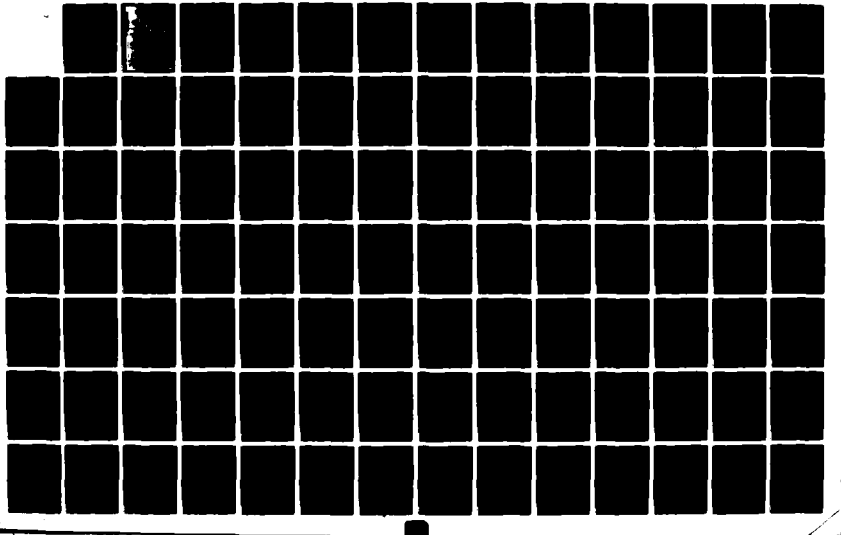
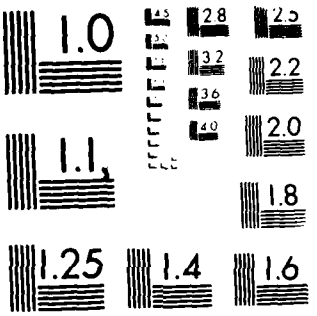
AD-A142 515 PARALLELISM IN MANIPULATOR DYNAMICS REVISION(U) 1/2
MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB R H LATHROP DEC 83 AI-TR-754-REV
UNCLASSIFIED N00014-80-C-0505 F/G 6/4 NL





Parallelism in Manipulator Dynamics

Richard H. Lathrop

MIT Artificial Intelligence Laboratory

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-754	2. GOVT ACCESSION NO. AD-A147515	3. RECIPIENT'S CATALOG NUMBER
4. TITLE *(and Subtitle)* Parallelism in Manipulator Dynamics		5. TYPE OF REPORT & PERIOD COVERED technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Richard H. Lathrop		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0505
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE December 1983 13. NUMBER OF PAGES 109
14. MONITORING AGENCY NAME & ADDRESS*(If different from Controlling Office)* Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. *(of this report)* UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

None

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

robots
robotics
industrial robots
cybernetics
parallel processing
pipeline processing
large scale integration

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This paper addresses the problem of efficiently computing the motor torques required to drive a lower-pair kinematic chain (e.g., a typical manipulator arm in free motion, or a mechanical leg in the swing phase) given the desired trajectory;i.e., the Inverse Dynamics problem. It investigates the high degree of parallelism inherent in the computations, and presents two "mathematically exact" formulationsespecially suited to high-speed, highly parallel implementations using special-purpose hardware or VLSI devices. In principle, the (cont. on back)

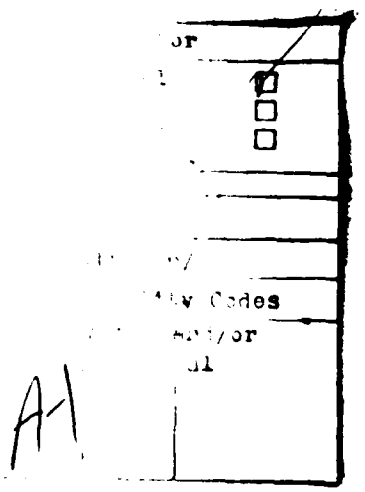
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

Block 20 cont.

formulation should permit the calculations to run at a speed bounded only by I/O. The first presented is a parallel version of the recent linear Newton-Euler recursive algorithm. The time cost is also linear in the number of joints, but the real-time coefficients are reduced by almost two orders of magnitude. The second formulation reports a new parallel algorithm which shows that it is possible to improve upon the linear time dependencey. The real time required to perform the calculations increases only as the $[\log_2]$ of the number of joints. Either formulation is susceptible to a systolic pipelined architecture in which complete sets of joint torques emerge at successive intervals of four floating-point operations. Hardware requirements necessary to support the algorithm are considered and found not to be excessive, and a VLSI implementation architecutre is suggested. We indicate possible applications to incorporating dynamical considerations into trajectory planning, e.g. it may be possible to build an on-line trajectory optimizer.



PARALLELISM IN MANIPULATOR DYNAMICS

by

RICHARD HAROLD LATHROP

ABSTRACT

This paper addresses the problem of efficiently computing the motor torques required to drive a lower-pair kinematic chain (e.g., a typical manipulator arm in free motion, or a mechanical leg in the swing phase) given the desired trajectory; i.e., the Inverse Dynamics problem. It investigates the high degree of parallelism inherent in the computations, and presents two "mathematically exact" formulations especially suited to high-speed, highly parallel implementations using special-purpose hardware or VLSI devices. In principle, the formulations should permit the calculations to run at a speed bounded only by I/O. The first presented is a parallel version of the recent linear Newton-Euler recursive algorithm. The time cost is also linear in the number of joints, but the real-time coefficients are reduced by almost two orders of magnitude. The second formulation reports a new parallel algorithm which shows that it is possible to improve upon the linear time dependency. The real time required to perform the calculations increases only as the $\lceil \log_2 \rceil$ of the number of joints. Either formulation is susceptible to a systolic pipelined architecture in which complete sets of joint torques emerge at successive intervals of four floating-point operations. Hardware requirements necessary to support the algorithm are considered and found not to be excessive, and a VLSI implementation architecture is suggested. We indicate possible applications to incorporating dynamical considerations into trajectory planning, e.g. it may be possible to build an on-line trajectory optimizer.

This report is a revision of a thesis submitted to M.I.T. in partial fulfillment of the requirements for the degrees Master of Science and Electrical Engineer.
Thesis Supervisor: Marvin Minsky
Title: Donner Professor of Science

Keywords: Robots, Robotics, Industrial Robots, Cybernetics, Parallel Processing, Pipeline Processing, Large Scale Integration

To my Parents —

and to All who have

Taught me of Life

Acknowledgements

I would like to thank all of the many people who have helped in so many ways with this effort — either through sharing technical expertise, ideas, and critiques; or by lending personal support, encouragement, and perspective.

Gratitude to those personally important is best personally expressed; but it is fitting to here acknowledge my tremendous intellectual debt to those who have contributed to my scientific, technical, and professional development. It would be impossible to list these many people in so short a space; those who have contributed most immediately and directly to this document are cited here, without prejudice to many other important influences.

Primary thanks are due to my thesis advisor, Marvin Minsky, for first exposing me to issues of parallelism in Artificial Intelligence, for having interesting ideas about it, and for supervising the research which originally led to the formulations described within these pages. Exposure to his insights has influenced me greatly and will doubtless continue to do so. John Hollerbach and Mike Brady read earlier drafts of this work, and I am very much indebted to them for many valuable suggestions on these and other issues. Their advice and encouragement was very important to my pursuit of this research. In particular, the patient urging and good counsel of John Hollerbach was indispensable and greatly appreciated; these pages reflect many of his helpful suggestions. Bob Giansiracusa was instrumental in spurring my early interest in the field, and in many long conversations has explored possible connections with a number of different areas. Jon Allen first taught me, and taught me well, about VLSI; a domain which gives substance to many otherwise speculative notions of parallelism. Thanks also to Bill Barrett and Bob Rogers for a stimulating forum and many rousing discussions on how best to capture the potential of VLSI by expressing complexity concisely.

I would also like to thank the M.I.T. Artificial Intelligence Laboratory for providing superb computing resources and an environment which is intellectually stimulating, and the National Science Foundation for providing support through a Graduate Fellowship.

This report is a revision of a thesis submitted to M.I.T. in partial fulfillment of the requirements of the degrees Electrical Engineer and Master of Science. It describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505. Personal support for the author was provided by a Graduate Fellowship from the National Science Foundation.

Table of Contents

0. OVERVIEW

The Inverse Dynamics problem consists (loosely) of computing the motor torques necessary to drive a mechanical manipulator through a specified motion: given that you know where it should go, compute what you have to do to get it to go there. Recently, efficient recursive formulations have been developed using both Newton-Euler and Lagrangian dynamics [16][29]. These have reduced the number of additions and multiplications (for n joints) from an $\mathcal{O}(n^4)$ dependency (see Table O.1) to one linear in the number of joints, $\mathcal{O}(n)$, requiring

$$(150n - 48)\ Mults + (131n - 48)\ Addns \qquad \text{(Newton-Euler)}$$

multiplications ($Mults$) and additions ($Addns$) when performed serially.

This paper investigates the high degree of parallelism inherent in the calculations, and presents two formulations especially suited to highly parallel implementations using special-purpose hardware or VLSI devices. Table O.1 shows the improvement over serial implementations. (Note that this reflects the algorithmically indicated cost, as in Hollerbach[16]; see the discussion at the beginning of Section III.)

The first formulation is again linear in the number of joints, but reduces the real-time coefficients by almost two orders of magnitude to

$$(2n + 3)\ Mults + (6n + 7)\ Addns \qquad \text{(Newton-Euler)}$$

The second formulation shows that by exploiting a novel parallel algorithm developed below, the time required to perform the calculations increases only as $\mathcal{O}(\log(n))$. The time dependencies are

$$(2\lceil \log_2(n+1) \rceil + 5)\ Mults + (6\lceil \log_2(n+1) \rceil + 10)\ Addns \qquad \text{(Newton-Euler)}$$

Either formulation is susceptible to a systolic pipelined architecture. We show below that the basic time cycle of the algorithm is 1 $Mult$ + 3 $Addns$. Thus, after the first complete set of joint torques had emerged from the pipeline, successive sets would appear at intervals

of four floating-point operations (4 *Flops*). This yields the ability to rapidly and efficiently evaluate a large number of alternatives.

Section I contains a brief introduction to the problem and review of previous work in the area.

Section II explains in detail the notation used in this paper . It is essentially an adaptation of the notation used by Hollerbach[16] and Luh et al.[29], which in turn derives from the Denavit-Hartenberg[10] convention for lower-pair linkages through Uicker[43] and Kahn[19]. The reader already broadly familiar with this notation should at least review Table II.1 where the notation is summarized.

Section III explicates the first approach considered, yielding an $\mathcal{O}(n)$ formulation with greatly reduced coefficients. Luh et al.[29] give the recursive form of the Newton-Euler formulation as shown in Table I.1. Many of the computations associated with any given joint (node) may proceed concurrently. For example, the computations of the variables ω_i (denoted by (*) in Table I.1) and $\dot{\omega}_i$ (**) do not interact (given that ω_{i-1} and $\dot{\omega}_{i-1}$ have already been computed) and hence may be performed at the same time by different sub-processors. Additionally, different sub-expressions of the same variable may often be computed concurrently by different sub-processors. Finally, by locally pipelining the recursive variables additional speed may be gained; e.g., the computation of $\dot{\omega}_{i+1}$ may be started before the computation of $\dot{\omega}_i$ has finished. (See Tables III.1 and III.2). Essentially, this formulation arises from trying to compute as much as possible as early as possible, and still remains within a basic linear structure.

Section IV shows the derivation of an $\mathcal{O}(\log(n))$ time dynamics formulation. This arises from restructuring the fundamental framework within which computation proceeds, together with a corresponding revision to the recursive equations. The linear recursive algorithm is, conceptually, a formalism for beginning at the manipulator base and propagating desired motion outward link by link to the tip, then propagating tip environmental forces and torques back inward link by link to the base (determining the needed joint motor torques along the way). (See Figure III.1). In contrast, the logarithmic recursive algorithm is a mechanism for recursively propagating desired motion (or, forces and torques) between any two adjacent groups of links. Conceptually speaking, the propagation of desired motion from base to tip

is accomplished by grouping together adjacent links on the first step to form $(n/2)$ groups of two links each, then on each succeeding step grouping adjacent pairs of groups together until after $\lceil \log_2(n) \rceil$ steps there is one group encompassing all links (actually, the intermediate groups are also formed). It is analogous to summing n numbers in $\lceil \log_2(n) \rceil$ steps by adding together first adjacent pairs, then adjacent pairs of pairs, and so forth. (See Figures IV.1 and IV.2).

A synthesis of the two approaches is presented in Section V. The techniques of Section III for exploiting parallelism within a node are applied to the $\mathcal{O}(\log(n))$ time structure of Section IV, yielding an $\mathcal{O}(\log(n))$ formulation with reduced coefficients.

Ultimately the algorithm must be expressed in hardware, and Section VI addresses a few words to potential implementations. The principle thrust of this paper lies in the analytic formulations above, and we will consider hardware only to the extent of showing that physical implementations are reasonable and feasible. We consider the optimal number of processors, buffering of intermediate results, and internal communication, only to the extent of showing that the requirements are reasonable.

Construction of suitable hardware using today's technology argues for a special-purpose VLSI chip, and Section VII presents one architecture suitable for this purpose. A primitive module consisting of two multipliers, one adder, and some registers is sufficient to support all of the computation required. Several such primitive modules may then be assembled, together with a suitable control structure, to produce a matrix-vector arithmetic module. These may then be connected into a network corresponding to the communication structure of the algorithm, and each module programmed by the host computer to execute the operation and communication sequencing necessary to implement the algorithm (or for that matter, any other high-speed straight-line matrix-vector computation).

Finally, Section VIII will very briefly allude to, without discussing in depth, a few possible extensions to this work. The ability to efficiently pipeline implies that a number of considered variations on the same basic manipulator trajectory could be explored in parallel, and the one having the most satisfactory dynamical characteristics for actual execution chosen from among that set. It may even be possible to build an on-line "optimizing trajectory compiler" in which the desired motion (trajectory) for the next several time periods is pre-planned,

the motor torques automatically generated, and the time sequence inspected slightly *before* the manipulator has actually arrived at the trajectory points, thereby incorporating some dynamical considerations into trajectory planning. Poggio has a result indicating that neural structures could perform an arithmetic multiplication in about a millisecond, which implies that in principle it would be possible to compute the Inverse Dynamics in approximately real time using a suitable neural structure. We close with a few remarks concerning generalization of the $\mathcal{O}(\log(n))$ embedding to other recursive algorithms.

Table O.1 — Comparison of Dynamics Formulations*

(adapted from Hollerbach[16])

O.1a — Comparison of Time Dependencies		
Method	Multiplications	Additions
Uicker/Kahn (original Lagrangian)	$32\frac{1}{2}n^4 + 86\frac{1}{2}n^3 + 171\frac{1}{4}n^2 + 53\frac{1}{3}n - 128$	$25n^4 + 66\frac{1}{3}n^3 + 129\frac{1}{2}n^2 + 42\frac{1}{3}n - 96$
Waters (partially recursive)	$106\frac{1}{2}n^2 + 620\frac{1}{2}n - 512$	$82n^2 + 514n - 384$
Hollerbach (4x4 Lagrangian)	$830n - 592$	$675n - 464$
Hollerbach (3x3 Lagrangian)	$412n - 277$	$320n - 201$
Luh, Walker, Paul (Newton-Euler)	$150n - 48$	$131n - 48$
Horn, Raibert (table look-up)	$2n^3 + n^2$	$n^3 + n^2 + 2n$
Luh, Lin (scheduled parallel N.E.)	$57n - 18$ *(estimated — see text)*	$50n - 18$ *(estimated — see text)*
Lathrop (linear parallel N.E.)	$2n + 3$	$6n + 7$
Lathrop (logarithmic parallel N.E.)	$2\lceil \log_2(n+1) \rceil + 5$	$6\lceil \log_2(n+1) \rceil + 10$
Lathrop (systolic pipeline)	1 *(successive — see text)*	3 *(successive — see text)*

** This table reflects the algorithmically indicated cost for the fully general 6-link rotary manipulator, as in Hollerbach[16]. By considering special cases, introducing configuration or workspace assumptions, or tailoring the computation, additional reductions are possible. See the discussion at the beginning of Section III.*

Table O.1 — Comparison of Dynamics Formulations*

(adapted from Hollerbach[16])

O.1b — Comparison for $n = 6$		
Method	Multiplications	Additions
Uicker/Kahn (original Lagrangian)	66,271	51,548
Waters (partially recursive)	7,051	5,652
Hollerbach (4x4 Lagrangian)	4,388	3,586
Hollerbach (3x3 Lagrangian)	2,195	1,719
Luh, Walker, Paul (Newton-Euler)	852	738
Horn, Raibert (table look-up)	468	264
Luh, Lin (scheduled parallel N.E.)	323 *(estimated)*	280 *(estimated)*
Lathrop (linear parallel N.E.)	15	43
Lathrop (logarithmic parallel N.E.)	11	28
Lathrop (systolic pipeline)	1 *(successive)*	3 *(successive)*

** This table reflects the algorithmically indicated cost for the fully general 6-link rotary manipulator, as in Hollerbach[16]. By considering special cases, introducing configuration or workspace assumptions, or tailoring the computation, additional reductions are possible. See the discussion at the beginning of Section III.*

1. INTRODUCTION

In active articulated mechanisms, including both artificial and biological systems, the parameters which one typically *can* directly control are the forces (in translational joints, sometimes called prismatic) and torques (rotational joints, sometimes called revolute) applied by the actuators to the joints. Unfortunately, the parameters in which one is frequently interested are the linear and rotational accelerations (hence also, velocities and positions). This gives rise to two dual problems; both highly complex and non-linear, and both desirable to calculate in real time.

The Direct (or Integral) Dynamics problem is to compute the mapping $\mathcal{D}$ from a set of applied joint forces and torques (τ_i, *arising from stimulation of the actuators*) into the resulting linear and rotational joint motions (accelerations $\ddot{q}_i$):

$$\mathcal{D}:\{\tau_i\} \mapsto \{\ddot{q}_i\}.$$

Computing such a mapping is equivalent to simulating the motion of the mechanism under the applied actuator effects. In this case one knows what one does to the thing, and wishes to find out where it will go in response.

The Inverse Dynamics problem is to compute the inverse of the above mapping; given the accelerations desired, find the forces/torques necessary:

$$\mathcal{D}^{-1}:\{\ddot{q}_i\} \mapsto \{\tau_i\}.$$

Given that one knows where one wants the thing to go, what does one have to do to make it go there? This is the question that the Inverse Dynamics seeks to answer. "Where one wants it to go" is the desired trajectory, the manipulator configuration as a function of time. The configuration may be completely specified by the joint positions, so the trajectory may be given by stating *each joint position as a function of time* (*i.e.*, $q(t)$, where q is an n-dimensional vector giving the actual positions q_i of the n joints). These functions are assumed to be twice time-differentiable to provide joint velocities and accelerations ($\dot{q}(t)$, $\ddot{q}(t)$).

The question is usually posed by giving the joint positions and velocities ($q(t_0)$, $\dot{q}(t_0)$) which describe the state of the manipulator at a given point in time (say, t_0), together with

the joint accelerations which are desired at that point ($\ddot{q}(t_0)$). The answer expected is the n motor torques (the n-dimensional vector $\tau(t_0)$ giving the motor torques $\tau_i(t_0)$ at each joint i) which, if applied to the manipulator at that point in its state-space, would induce the desired accelerations. Typically one measures q and $\dot{q}$, while $\ddot{q}$ is supplied by a higher-level planning and control module. Position and velocity are the time integrals of acceleration, so acceleration control suffices, at least in a "mathematically exact" sense. There are a host of practical problems which insure that the model and the reality it models never quite match, and which we shall relegate to feedback. For control purposes the formalism provides a good first approximation to the "inverse plant", which is close enough to render feedback correction feasible.

Computing the motor torques is quite complicated, however, due to the high degree of non-linearity inherent in rigid-body rotational mechanics. The torques supplied must compensate for the inertia of the manipulator, gravitational force, the Coriolis and centrifugal forces, and viscous friction at the joints. Viscous frictional forces often depend only on q_i and $\dot{q}_i$ at joint i; hence they are susceptible to relatively simple correction and will therefore be ignored. All of the other terms vary in a non-linear fashion depending on the manipulator configuration at a given point in time; additionally, the Coriolis and centrifugal forces also depend on all pairwise products $(\dot{q}_i\dot{q}_j,\ 1 \leq i,j \leq n)$ of joint velocities.

This complicated computation has until recently posed a bottleneck in on-line control of manipulators, and much effort has been expended in devising more time-efficient methods. Typical resonant frequencies of many mechanical manipulators is around $10Hz$, so the computation must be repeated at about $60Hz$ or faster[29]. Uicker[43] and Kahn[19] derived an early formulation for an n-link manipulator based on the Lagrange equations. This had an $\mathcal{O}(n^4)$ time complexity and required 7.9 seconds on a PDP 11/45 to compute the torques for just one point in the trajectory[29]. This was too slow for on-line control. Efforts to improve this time have either explored other computational algorithms [44],[45],[35],[30],[29],[16], made simplifying assumptions [4],[35], or substituted table look-up for computation [38],[1],[2].

Since only the Coriolis and centrifugal terms involve pairwise products of all joint velocities, a common simplification is to just ignore them. Unfortunately this works well only

at low velocities where the product terms are small. During fast motion the Coriolis and centrifugal terms may dominate the computation [29] to such an extent that attempts to control the errors by feedback require excessive corrective torques[38].

Table look-up is in principle the fastest method of obtaining the necessary torques. If one could index a table of size $O(n^3)$ using the $3n$ values of $(q_i, \dot{q}_i, \ddot{q}_i, 1 \leq i \leq n)$, all computation could be replaced by memory references. This extreme was explored by Albus[1],[2]. Raibert[37] reduced the table size by proposing a table indexed by position and velocity (with acceleration handled separately), and this was further refined by Raibert and Horn[38] to a table indexed only by position. Nonetheless for fine enough division of the table dimensions the size required remains enormous, filling the table and interpolating between stored values present problems, and the table is valid only for one particular load (e.g., mass of object grasped)[16].

Two other formulations, not reflected in Table O.1 because particularized to special cases and hence not directly comparable, deserve mention. Kane and Levinson[20] discuss a formalism (Kane's Dynamics, originally developed for complex spacecraft control) based on generalized coordinates and velocities similar to the Lagrangian approach. The dynamical parameters can be represented explicitly so as to exploit simplifications arising from manipulator configuration or workspace constraints, though the computational complexity therefore reflects both configuration and workspace assumptions. They analyze the Stanford arm under the assumption that the workspace never requires the second joint to approach $\theta_2 = 0°$ or $\theta_2 = 180°$ (due to numerical instabilities there). Hollerbach and Sahar[18] present a method of merging the inverse kinematics with the inverse dynamics, particularized to the case of a robot with a spherical wrist. Many of the kinematic parameters generated in the inverse kinematics are needed in the inverse dynamics, and additional savings arise from the simplification of assuming a spherical wrist. Other special cases are discussed in the beginning of Section III.

The most successful formulations of the general inverse dynamics involve recursive algorithms. Waters[44] first presented an $O(n^2)$ partial recursive form of the Lagrange equations, which was made fully linear recursive by Hollerbach[16]. Hollerbach also contains an overview of contrasting approaches to the inverse dynamics problem, to which the

interested reader is referred for further details. Orin et al.[34] first presented a linear recursive form of the Newton-Euler equations, which was refined by Luh et al.[29].

We have considered both the Newton-Euler and the Lagrangian formulations, in the form presented by Luh et al.[29] and Hollerbach[16]. Silver[40] has shown them to be fundamentally equivalent, differing mainly in that the representation of angular velocity in the Newton-Euler equations is more efficient. Reflecting this, the parallel formulations for both formalisms are found to require approximately equal parallel time to compute, but the Lagrangian formulation would require substantially more hardware to implement. This paper will thus present only the Newton-Euler results. Complete details for the Lagrangian case may be found in Lathrop[25].

The underlying intuition in both cases is the following. The motion (by which we will generally mean: position, velocity, and acceleration) of the manipulator base is assumed known, as is the motion of each individual joint. By beginning at the base and accounting for the (known, desired) joint motion at each joint, the motion of each successive link may be cascaded recursively from the base of the manipulator to the tip. Since the forces and torques applied by the environment to the last manipulator link (the tip, workhead, or end-effector) are known or measured, and the motion of the last link is known, the force or torque at the last joint necessary to drive the last link through its desired motion may be calculated directly from free-body rotational mechanics. This in turn allows calculation of the forces and torques transmitted across the last joint to the next inboard neighbor link, which in turn allows calculation of the next but last force or torque from rotational mechanics. Continuing in toward the base in this fashion, and accounting for the forces and torques passed across each joint, the force or torque necessary at each joint to drive each link through its desired motion may be calculated in linear time. The required motor force or torque at each joint is then the component of force or torque along or about the joint axis. Thus in linear time, the motor torques needed to support a desired motion may be computed.

A number of practical attempts have been made to relieve the host computer of some of the computational burden associated with manipulator control. The principle idea is that machine management should be performed outside the main computer (CPU). A common

strategy is to use a single microprocessor to control the robot, and the host to control the controller, e.g. [11],[13], and [23]. This paper will not consider these further because the parallelism achieved is minimal, and they do not address arm dynamics.

Most attempts to apply parallel processing to manipulator control have involved using microprocessors to servo individual joints, and have not attempted to include dynamical considerations. Typically, this involves the microprocessor as the active element in a joint control feedback loop with sensors to monitor the error (usually of joint position). Also, the individual joint servos for each joint axis are usually under the control of a master microprocessor, which coordinates their actions with commands from the host. In fact, usually the servo microprocessors will not communicate with each other directly at all. Shin and Malin[39] discuss one control strategy for this general approach.

Acting under this general paradigm, Cook et al.[9] discuss a configuration of seven 68000s and a programmable logic controller, designed for welding underwater pipelines. Kuo[24] describes an arm mounted on a mobile platform, having one microprocessor per motor and based on an AIM-65 system. Rafauli et al.[36] control a modified Unimate 2000, using one Intel 8748 to servo each axis based on positional feedback and a master composed of an iSBC 86/12A combined with an iSBC 337. Gupta[14] advocates several advantages of the MK68000 for similar applications. A single-board controller for a pneumatic drive arm, using one microprocessor at each axis with feedback from air pressure and speed as well as position, is presented by Goshorn[12]. Carlisle[8] additionally dedicates several microprocessors to sub-tasks such as sensor monitoring or I/O, though his interest leans somewhat more to computer numerically controlled machines. Distributed manipulator control schemes for servo-manipulators used in nuclear reactor maintenance are described by Besant[5] and Martin et al.[31]. The OSU Hexapod (an 18-degree-of-freedom, motor-driven walking machine) is controlled by an experimental multiprocessor consisting of five LSI-11s [21]. These are reconfigurable so that tree, star, and loop structures can be simulated. This multiprocessor has also been used for real-time optimization of leg tip forces, a task which is not strictly parallel in nature.

Mudge and co-workers in several papers outline a scheme whereby the general purpose computer incorporates attached special-purpose processors for real-time numerically inten-

sive computations. The special processor proposed is a single chip implementation, which would interpolate between set points from the host and compute the correction torques for each joint of the robot arm, replacing their current PUMA control scheme of one LSI-11/02 and six 6503s [26]. Though this particular application ignores dynamic coupling between joints, another paper [33] proposes a scheme to unify Resolved Motion, Gross Motion, and Fine Motion, based on the Newton-Euler formalism, suitable for implementation in real-time on their processor. The processor (called by them NP, the Numerical Processor) is described [32] as lying between Floating Point Systems' AP120B (a high performance numerically oriented attached processor) and the Intel 8087 (a single chip numerically oriented attached processor in the Intel 8087 family). A similar approach is described by Turner et al.[42] involving a distributed processing architecture using three microcomputers in a pipelined configuration, also proposed for a broad class of advanced manipulator control algorithms.

In the work most closely related to this paper, Luh and Lin describe a procedure for scheduling the sub-tasks of a group of microprocessors computing the Newton-Euler dynamics [27],[28]. One microprocessor is again assigned to each controlled joint axis; but in contrast to the servo-based approaches above, the microprocessors do communicate and arm dynamics are explicitly computed. Each microprocessor computes the recursion variables which correspond to its joint axis. Since these variables (as well as intermediate partial results) recursively depend on each other in various different ways, often some microprocessor(s) will be idle while waiting for others to complete pending sub-tasks and idle time must be included in the schedule. It is a non-trivial scheduling problem to assign each sub-task of each microprocessor a specific execution sequence which minimizes the global computation time. Because the form of the equations (and hence the sub-tasks to be performed) for a joint differs depending on whether that joint is rotational or translational, in general each new manipulator configuration requires a new rescheduling of sub-tasks.

The procedure adopted is a modified branch-and-bound search through the space of possible sub-task orderings, terminating when the minimum-time ordering has been found. Any feasible ordering which accomplishes all sub-tasks is first found, then refined by generating and comparing alternatives (other, partial, orderings from branch (choice) points). The estimated total time of a partial task ordering is its partial time so far (including

idle time), plus the time for all its remaining sub-tasks to complete if idle time is ignored. Since this is guaranteed to be an underestimate of true total time, branch-and-bound search applies. Each partial ordering is extended until either its estimated total time exceeds that of the minimum-time feasible ordering so far found, or it is extended to a complete feasible ordering of less total time and so becomes the new minimum. At the conclusion of this procedure, the path of minimum true total time is the optimum schedule. Though by its nature this procedure is not subject to precise analysis of the computational complexity of the resulting schedules, the authors report a concurrency factor of 2.64 on the Stanford arm. This estimated factor is used in Table O.1.

Table I.1 — Linear Recursive Newton-Euler Formulation
(after Luh et al. [29])

Newton-Euler Backward Recursion:

$$\omega_i = \begin{cases} A_i^T(\omega_{i-1} + z_{i-1}\dot{q}_i) & \text{if joint } i \text{ rotational;} \\ A_i^T\omega_{i-1} & \text{if joint } i \text{ translational.} \end{cases} \quad (*)$$

$$\dot{\omega}_i = \begin{cases} A_i^T(\dot{\omega}_{i-1} + z_{i-1}\ddot{q}_i + \omega_{i-1} \times z_{i-1}\dot{q}_i) & \text{if joint } i \text{ rotational;} \\ A_i^T\dot{\omega}_{i-1} & \text{if joint } i \text{ translational.} \end{cases} \quad (**)$$

$$\ddot{p}_i = \begin{cases} A_i^T\ddot{p}_{i-1} + \dot{\omega}_i \times p_i^* + \omega \times (\omega_i \times p_i^*) & \text{if joint } i \text{ rotational;} \\ A_i^T\ddot{p}_{i-1} + \dot{\omega}_i \times p_i^* + \omega \times (\omega_i \times p_i^*) + A^T z_{i-1}\ddot{q} + 2\omega_i \times A_i^T z_{i-1}\dot{q}_i & \text{if joint } i \text{ translational.} \end{cases}$$

$$\ddot{r}_i = \omega_i \times (\omega_i \times r_i^*) + \dot{\omega}_i \times r_i^* + \ddot{p}_i$$

$$F_i = m_i\ddot{r}_i$$

$$N_i = J_i\dot{\omega}_i + \omega_i \times (J_i\omega_i)$$

Newton-Euler Forward Recursion:

$$f_i = F_i + A_{i+1}f_{i+1}$$

$$n_i = A_{i+1}n_{i+1} + N_i + s_i^* \times F_i + p_i^* \times (A_{i+1}f_{i+1})$$

$$\tau_i = z_{i-1} \cdot n_i$$

2. NOTATION

The notation is based on that used by Hollerbach [16] and Luh et al. [29] in their analyses of the linear recursive inverse dynamics, which in turn derives from the Denavit and Hartenburg [10] convention for lower-pair chains. Coordinate systems are fixed in the body of each link and related rotationally by 3×3 matrix coordinate transforms and translationally by distinguished body-fixed position vectors. The notation ${}^{i}v_j$ is used to denote the vector v_j referred to coordinate system O_i.

It is worth emphasizing that, with few exceptions, *ALL* vectors in this paper are referred to link coordinates. This obviates the complexity and computational overhead of transforming everything to base coordinates. Elsewhere in the literature, ${}^{0}v_j$ (denoting the vector v_j referred to base coordinates) is usually abbreviated simply as v_j. Since we will almost never refer any vector to other than its own link coordinates, we adopt instead the notation-simplifying convention that v_j abbreviates ${}^{j}v_j$, a vector referred to its own coordinate system O_j.

The links of a mechanism are numbered consecutively 1 to n from base to tip, with link 0 denoting the base reference frame. There is another fictitious link $n+1$ attached to the tip when convenient, which may represent the object grasped or account for environmentally applied forces and torques at the workpiece. Attached to link i is a right-handed orthogonal coordinate system O_i with axes (x_i, y_i, z_i).

Joints (equivalently, hinges) occur between the links. Each joint is denoted by the number assigned to its distal (outboard) link, so that joint i connects links $i-1$ and i. Joints may be either rotational or translational, but may have only one degree of freedom. Multiple degrees of freedom at a joint are modeled by introducing fictitious links having zero mass and length.

For any two adjacent coordinate systems O_i and O_{i-1} there is an orthonormal rotation matrix A_i mapping vectors whose coordinates are referenced to O_i into the corresponding vectors referenced to O_{i-1}. Note that this is a pure rotation. The coordinate systems are located in the links so as to simplify the form of this matrix. The orthonormal basis vectors of O_i are arranged as follows:

z_i lies along the positive axis of joint $i+1$,
x_i lies along the common normal from z_{i-1} to z_i,
$y_i = z_i \times x_i$ completes the right-handed orthonormal system.

Since the joints have but one degree of freedom, x_i, z_i, and z_{i-1} are all body-fixed vectors in $\mathcal{O}_i$ (i.e., have a fixed direction relative to the body-fixed coordinate system). The rotational orientation of two adjacent systems is completely described by

α_i is the angle between z_{i-1} and z_i in a right-handed sense about x_i,
θ_i is the angle between x_{i-1} and x_i in a right-handed sense about z_{i-1}.

The vectors z_{i-1} and z_i being both fixed in $\mathcal{O}_i$, α_i is constant. Since z_{i-1} lies along the joint axis of joint i, and x_{i-1} and x_i are fixed in $\mathcal{O}_{i-1}$ and $\mathcal{O}_i$ respectively and both normal to z_{i-1}, θ_i measures the relative rotation about the joint axis between the two systems. Thus if joint i is rotational then θ_i is the joint variable, otherwise θ_i is constant (see Figure II.1).

By the preceding remarks, the rotation matrix A_i corresponds to a coordinate rotation about x_i by an angle α_i (which aligns z_i and z_{i-1}) followed by a rotation about the rotated z_i ($= z_{i-1}$) by an angle θ_i (which aligns the other two pairs of basis vectors). If the first rotation is represented by the matrix Φ_i and the second by Θ_i, then

$$\Phi_i = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i \\ 0 & \sin\alpha_i & \cos\alpha_i \end{bmatrix}$$

$$\Theta_i = \begin{bmatrix} \cos\theta_i & -\sin\theta_i & 0 \\ \sin\theta_i & \cos\theta_i & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$A_i = \Theta_i \Phi_i$$

$$= \begin{bmatrix} \cos\theta_i & -\sin\theta_i \cos\alpha_i & \sin\theta_i \sin\alpha_i \\ \sin\theta_i & \cos\theta_i \cos\alpha_i & -\cos\theta_i \sin\alpha_i \\ 0 & \sin\alpha_i & \cos\alpha_i \end{bmatrix}$$

The translational orientation of two adjacent systems is completely described by

a_i is the distance between the origins of $\mathcal{O}_{i-1}$ and $\mathcal{O}_i$ measured along x_i;
s_i is the distance between x_{i-1} and x_i measured along z_{i-1}.

Since x_i, z_i, and z_{i-1} are fixed in $\mathcal{O}_i$, a_i is constant. z_{i-1} lies along the joint axis of joint i, so if joint i is translational s_i will be the joint variable, otherwise s_i is constant.

The following link i vectors, referenced to $\mathcal{O}_i$, permit a convenient specification of the translational relationship between adjacent coordinate systems

p_i^* a vector to origin $\mathcal{O}_i$ from $\mathcal{O}_{i-1}$

r_i^* a vector to the center of mass of link i from $\mathcal{O}_i$

s_i^* $= p_i^* + r_i^*$, a vector to the center of mass of link i from $\mathcal{O}_{i-1}$.

From the above remarks it is clear that

$$p_i^* = s_i A_i^T z_{i-1} + a_i x_i$$

but in virtue of the form of $A_i = \Theta_i \Phi_i$ and its constituents, we have

$$\begin{aligned} \Theta_i^I z_{i-1} &= z_{i-1} \\ A_i^I z_{i-1} &= \Phi_i^T \Theta_i^T z_{i-1} \\ &= \Phi_i^T z_{i-1} \end{aligned}$$

which is body-fixed in $\mathcal{O}_i$. Thus p_i^* is composed of a part $a_i x_i$ which is fixed in $\mathcal{O}_i$ and represents (constant) translation normal to both z_{i-1} and z_i, together with a part $s_i \Phi^I z_{i-1}$ whose direction is fixed in $\mathcal{O}_i$ and whose magnitude represents translation along z_{i-1} referred to $\mathcal{O}_i$. If joint i is rotational then s_i is constant and so therefore is p_i^*, else p_i^* incorporates the result of translational joint motion at joint i.

The rotation matrices A_i may be cascaded by defining the rotation matrix ^iW_j, which maps $^j u_m$ into $^i u_m$. Since the inverse of an orthonormal matrix is equal to its transpose it follows that

$$\begin{aligned} {}^iW_j &= ({}^jW_i)^{-1} = ({}^jW_i)^T \\ &= \begin{cases} A_{i+1} \cdots A_j & \text{if } i < j, \\ A_i^T \cdots A_{j+1}^T & \text{if } i > j, \end{cases} \end{aligned}$$

the superscript T denoting transpose in either matrices or vectors.

From the above it is clear that

$${}^0W_j = [({}^0x_j), ({}^0y_j), ({}^0z_j)]$$

and

$$
\begin{aligned}
{}^{i}W_j &= ({}^{0}W_i^T)\,{}^{0}W_j \\
&= ({}^{i}W_k)({}^{k}W_j) \\
&= [({}^{i}x_j), ({}^{i}y_j), ({}^{i}z_j)].
\end{aligned}
$$

In the sections on logarithmic recursion it will be necessary to introduce another notation for consistency with the formalisms developed there. We will use $W_{i,j}$ to denote the mapping from $\mathcal{O}_j$ to $\mathcal{O}_{i-1}$, so that

$$W_{i,j} = {}^{i-1}W_j.$$

Also in the sections on logarithmic recursion we will slightly abuse the dot notation for vector time differentiation (i.e., $\dot{\omega}_{i,i}$ and $\dot{p}_{i,i}$). Elsewhere in the literature this denotes differentiation in the inertial frame, with differentiation in a rotating frame indicated by $'$ or $\bullet$. We will take *ALL* terms $\dot{u}_{i,i}$ to denote differentiation of $u_{i,i}$ in $\mathcal{O}_{i-1}$. This becomes equivalent to the standard notation at $\dot{u}_{0,i}$, the case of interest, and eliminates proliferation of $'$ or $\bullet$ superscripts in much the same spirit that taking v_0 eliminated superscripts of 0. This is explained more fully in Section IV.

Table II.1 — Summary of Notation Used

We define the following:

m_j	the mass of link j,
r_j^*	a vector to the center of mass of link j from the origin O_j,
p_j^*	a vector to the origin O_j from the origin O_{j-1},
$\ddot{p}_j^g$	accounts for gravitational acceleration, g if $j = 0$, else 0,
s_j^*	$= p_j^* + r_j^*$, a vector to the center of mass of link j from the origin O_{j-1},
n_j^*	$= r_j^*/m_j$,
ω_j	the angular velocity vector of link j,
J_j	the inertial tensor (with respect to its center of mass) of link j,
q_j	the joint generalized variable for joint j, θ if rotational and s if translational,
τ_j	the joint generalized actuator force at joint j, torque if rotational and force if translational,
F_j	the total force (excluding gravity) on link j,
N_j	the total torque on link j,
f_j	constraint force (unknown) exerted on link j by link $j-1$,
n_j	constraint torque (unknown) exerted on link j by link $j-1$,
A_j	a pure rotation matrix mapping vectors in O_j into vectors in O_{j-1},
${}^iW_j$	a pure rotation matrix mapping vectors in O_j into vectors in O_i,
$W_{i,j}$	$= {}^{i-1}W_j$,
$\dot{u}_{a,b}$	$u_{a,b}$ differentiated in O_{a-1} and referred to O_b.

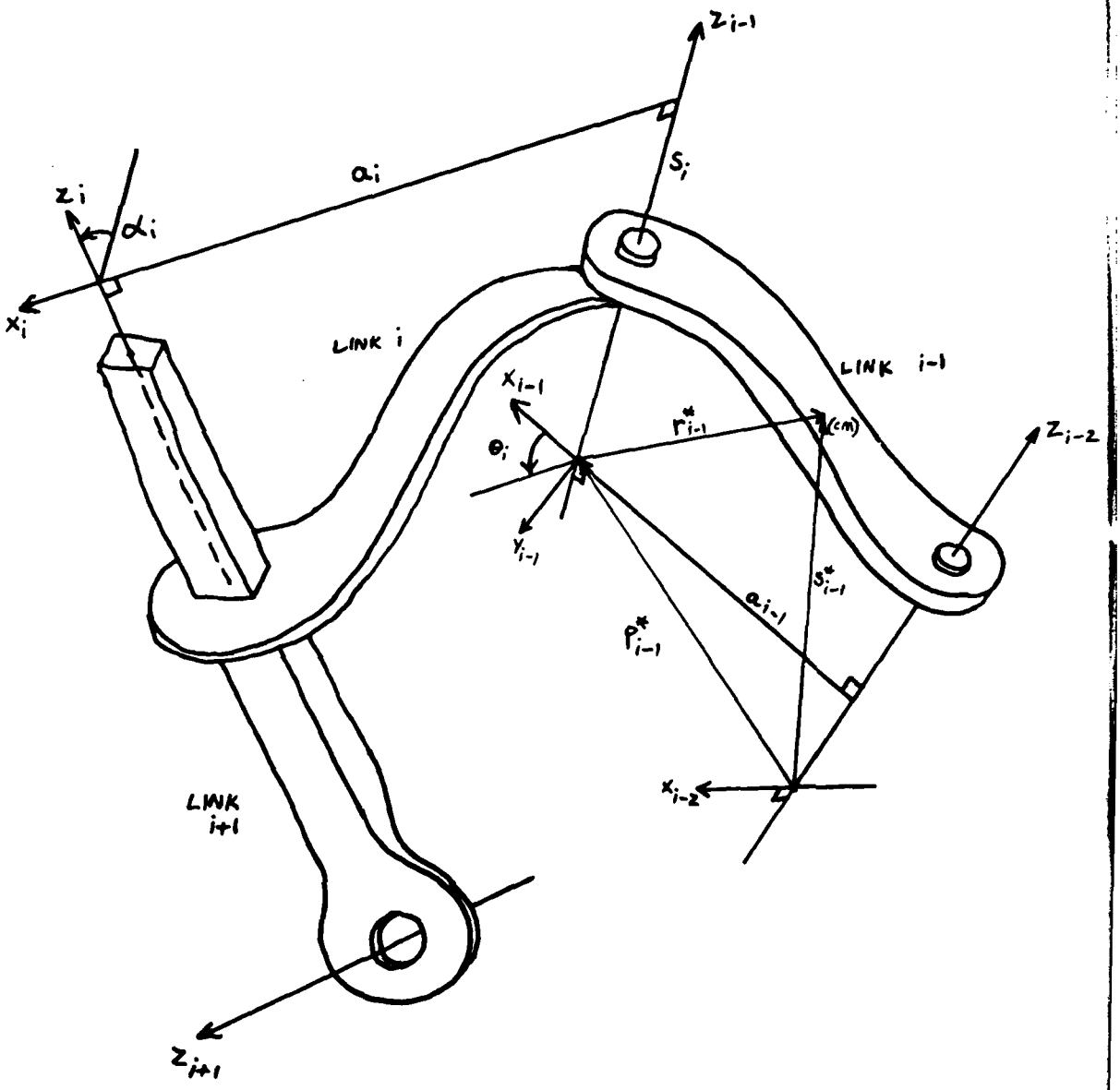


Figure II.1 — Notation of Manipulator Parameters

9. PARALLELISM WITHIN A NODE

This section will investigate the effect of exploiting parallelism within a node (joint), while remaining within the general linear recursive framework. As a conceptual aid in this section we assume that there is one group of parallel processors for each joint (node) on the forward and the backward recursion. However, since only one node is active in the computation of any given variable at any one step, and all nodes are identical, an implementation could be constructed using only one processor group by connecting the output back to the input through a buffer. Details of how this might actually be done are explored in Section VI. If only one processor group is used, computation of one set of joint torques must be completed before the next can begin. Otherwise, with one processor group for each joint (node), it is possible to systolically pipeline successive sets of joint torques at intervals of 1 *Mult* + 3 *Addns*, or 4 *Flops*.

The linear recursive structure of the Newton-Euler computations may be shown as a directed graph as in Figure III.1. The essential structure of the algorithm can be clearly seen — acceleration is propagated outward (incorporating at each stage the next joint acceleration) and forces are thereafter propagated inward (allowing calculation of joint torque at each stage). Nodes represent the total processing associated with each joint in the forward or backward recursion, and directed arcs represent data dependencies. It is clear that reducing the computational time incurred at each joint (node) would imply a linear reduction in the total computational time (a constant factor speed-up). For reasons which will be explained in Section IV, a double subscript is used in Figure III.1: thus $(0,j)$ on the backward recursion, and (j,n) on the forward, denotes the variables output by node j. Non-systolic and systolic arrangements are indicated in Figures III.2 and III.3.

Tables III.1 and III.2 show the detailed internal structure of each node of the directed graph of Figure III.1. The computation times given in Table III.1 reflect the times represented by the indicated operation. In general we follow Hollerbach[16] in accepting the principle that economies in computation should be explicit in the formulation rather than implicit in the programming, and Table O.1 reflects his analysis for the fully general 6-link rotary manipulator. It should be noted, however, that by "hand-tailoring" the computation to

account for special cases the computational cost can be frequently be reduced below that shown in Table O.1. For example, if the particular manipulator configuration is such that some of the constant angles α_i are multiples of $\frac{\pi}{2}$, then the factors $\cos\alpha_i$ and $\sin\alpha_i$ become 0 and ± 1 and multiplication by the rotation matrix A_i can be reduced from 9 *Mults* and 6 *Addns* to 4 *Mults* and 2 *Addns*. Since $y_i \cdot p_i^* = 0$, vector cross products involving p_i^* can be calculated with a special sub-routine in only 4 *Mults* and 2 *Addns*. Several other optimizations are possible, e.g. see [18],[20],[28],[29]. Many of these apply to the Stanford arm, which therefore has a substantially lower computational cost than the fully general case. Newton-Euler dynamics particularized to the Stanford arm requires only 308 *Mults* and 254 *Addns* [28], and Kane's dynamics requires only 646 *Mults* and 394 *Addns* [20]. A rotary manipulator particularized to a spherical wrist and non-spinning base requires only 448 *Mults* and 361 *Addns* [18]. Our analysis captures the fully general case with time bounds substantially below these, and in any event the intended implementation in VLSI argues strongly for a regular and systematic treatment which avoids all special cases.

The times shown in Tables III.1 and III.2 reflect, for each variable, the rotational case only. The extension to the translational case follows directly in analogous fashion, and the time bounds stated remain almost exactly the same with only minor variation in the constant term.

Note that, while on either the forward or backward recursion, the linear coefficient in the total time cost is determined by the time required to propagate the recursion variables through a single node. More specifically, this is the interval between the time that the recursion variables become available to one node and the time that they are made available to the next node. However, nothing constrains all of the variables to be made available at the same time. Since the different recursion variables are used at different times in the computation, relative delays are acceptable provided that each variable is available by the time that it is required in the computation.

If the different variables become available to a node at staggered times (and in turn are made available to the next node at equally staggered times), the linear time coefficient is determined by the longest time required to propagate any single variable across the node. In determining this time only ω_i, $\dot{\omega}_i$, $\ddot{p}_i$, f_i, and n_i need be considered, as the other variables

are not propagated down the recursive chain.

Examining Table III.1 for the Newton-Euler formulation, it is clear that (given the proper staggering constant offsets) no variable on either the forward or backward recursion requires longer than a matrix-vector multiplication and a vector addition to propagate through a node. This is evident because each propagated variable becomes available to the $(i+1)^{\underline{th}}$ node one matrix-vector multiplication and one vector addition after it is made available to the $i^{\underline{th}}$ node. Thus, the time required for the entire dynamics calculation may be reduced to

$$2n \cdot (MV + VA) + C$$

where (MV) and (VA) are respectively the time required to perform a matrix-vector multiplication and a vector addition, and C is a constant which accounts for initiating the calculation.

This is the case even though each individual variable may take longer than $(MV + VA)$ to compute. Table III.2 illustrates this arrangement graphically. The times when each of the variables become available are shown in Table III.2, as well as the intermediate partial results. For simplicity of presentation in Table III.2, a timing model is used in which $1\ Mult = 1\ Addn = 1\ Flop$. This metric will also be used to establish the relative times at which various partial results are computed and made available.

ω_i depends on nothing except ω_{i-1} and $z_{i-1}\dot{q}_i$, and requires time $(MV+VA)$ to compute once these are available. It is clear that successive values of ω_i will become available at intervals of $(MV + VA)$, since each depends upon nothing but its preceding recursive variable value and the input. $\dot{\omega}_i$ depends on ω_{i-1} as well as on $\dot{\omega}_{i-1}$ and $z_{i-1}\ddot{q}_i$. However, ω_{i-1} and $z_{i-1}\ddot{q}_i$ are required in the computation well before $\dot{\omega}_{i-1}$. Given the availability of $\{\omega_i\}$ at intervals of $(MV + VA)$, any ω_i-dependent intermediate partial results required by $\dot{\omega}_i$ can be calculated before $\dot{\omega}_{i-1}$ becomes available (by delaying $\dot{\omega}_1$ if necessary, as will be seen below). These intermediate results already computed, when $\dot{\omega}_{i-1}$ does become available it will be possible to compute $\dot{\omega}_i$ in time $(MV + VA)$. Meanwhile computational precursors to $\dot{\omega}_{i+1}$ have been similarly calculated, so that when $\dot{\omega}_i$ becomes available it is then possible to compute $\dot{\omega}_{i+1}$ in time $(MV + VA)$; and so forth. In this fashion, it can be seen that the availability of $\{\omega_i\}$ at intervals of $(MV + VA)$ implies the availability of

$\{\dot{\omega}_i\}$ at similar but offset intervals. By a similar reasoning process, both of these imply the availability of $\{\ddot{p}_i\}$ at intervals of $(MV + VA)$. The structure and timing which realize this are shown in Table III.2. Similar considerations would hold for the forward recursion.

The availability offsets and the constant C may both be determined from Table III.1. This is done in Appendix A, where the appropriate offsets are shown to be

$$Avail(\dot{\omega}_{i-1}) \geq Avail(\omega_{i-1}) + VC + VA \qquad (*)$$

$$\begin{aligned} Avail(\ddot{p}_{i-1}) &\geq Avail(\omega_{i-1}) + 2VC + 3VA \\ Avail(\ddot{p}_{i-1}) &\geq Avail(\dot{\omega}_{i-1}) + VC + 2VA. \end{aligned} \qquad (**)$$

$$Avail(n_{i+1}) \geq Avail(f_{i+1}) + VC + VA. \qquad (***)$$

The three equations above, (*), (**), and (***), define the constant offsets or relative delays by which the propagation of the Newton-Euler recursion variables should be staggered in order to achieve the stated time bound. Note that if computation is allowed to proceed on a scheme whereby a node operates on its inputs as soon as the data becomes available, then these offsets will be naturally set up and maintained by the inherent computational delays shown in Table III.1.

Next we determine the constant C by showing when τ_1, the last generalized joint force of the forward recursion, becomes available as output. This may also be done from Table III.1, and details are also shown in Appendix A. There it is shown that, assuming all input values become available simultaneously at time $t = 0$, the time required to calculate the Newton-Euler dynamics exploiting maximal linear parallelism is

$$2n \cdot (1\ Mult + 3\ Addns) + (5\ Mults + 9\ Addns).$$

Due to differences (not requiring additional computation by the host) in assumptions about the form of the input and output (not in the computation), this is slightly lower (by $2\ Mults + 2\ Addns$) than given in [25]. Specifically, we assume that the input buffer will deposit $\dot{q}_i$ and $\ddot{q}_i$ as the third scalar coordinate behind two pre-stored scalar zeroes so as to make available the vectors $z_{i-1}\dot{q}_i$ and $z_{i-1}\ddot{q}_i$ directly, and that the output buffer will directly return the third scalar coordinate of f_i (for translational joints) or n_i (for rotational joints) as the indicated joint force or torque. This input/output convention avoids the algorithmically indicated cost of SV and VD without introducing special cases into the computational structure.

Table III.1 — Relative Time of Linear Data Dependencies

Linear Backward Newton-Euler Recursion				
Var.	Waits On‡	Time at Step End	Step	Cost†
ω_i	$a = Avail(\omega_{i-1})$	*Input*	$T_1 = z_{i-1}\dot{q}_i$	$\bullet\bullet\bullet$
		$a + VA$	$T_2 = \omega_{i-1} + T_1$	VA
		$a + MV + VA$	$\omega_i = A_i^T T_2$	MV
$\dot{\omega}_i$	$a = Avail(\omega_{i-1})$	*Input*	$T_3 = z_{i-1}\ddot{q}_i$	$\bullet\bullet\bullet$
	$b = Avail(\dot{\omega}_{i-1})$	$a + VC$	$T_4 = \omega_{i-1} \times T_1$	VC
		$a + VC + VA$	$T_5 = T_3 + T_4$	VA
		$\max(b, a + VC + VA) + VA$	$T_6 = T_5 + \dot{\omega}_{i-1}$	VA
		$\max(b, a + VC + VA) + MV + VA$	$\dot{\omega}_i = A_i^T T_6$	MV
$\ddot{p}_i$	$c = Avail(\omega_i)$	$c + VC$	$T_7 = \omega_i \times p_i^*$	VC
	$d = Avail(\dot{\omega}_i)$	$c + 2VC$	$T_8 = \omega_i \times T_7$	VC
	$e = Avail(\ddot{p}_{i-1})$	$d + VC$	$T_9 = \dot{\omega}_i \times p_i^*$	VC
		$\max(d + VC, c + 2VC) + VA$	$T_{10} = T_8 + T_9$	VA
		$e + MV$	$T_{11} = A_i^T \ddot{p}_{i-1}$	MV
		$\max(e + MV, d + VC + VA, c + 2VC + VA) + VA$	$\ddot{p}_i = T_{10} + T_{11}$	VA
$\ddot{r}_i$	$c = Avail(\omega_i)$	$c + VC$	$T_{12} = \omega_i \times r_i^*$	VC
	$d = Avail(\dot{\omega}_i)$	$c + 2VC$	$T_{13} = \omega_i \times T_{12}$	VC
	$f = Avail(\ddot{p}_i)$	$d + VC$	$T_{14} = \dot{\omega}_i \times r_i^*$	VC
		$\max(d + VC, c + 2VC) + VA$	$T_{15} = T_{13} + T_{14}$	VA
		$\max(f, d + VC + VA, c + 2VC + VA) + VA$	$\ddot{r}_i = T_{15} + \ddot{p}_i$	VA

† *See end of table (continued next page).*

‡ *"$Avail(X_{i-1}) = t$" means that variable X_{i-1} is made available to the i^{th} node at time t (on the backward recursion; substitute X_{i+1} on the forward recursion).*

Table III.1 — Relative Time of Linear Data Dependencies (continued)

Linear Backward Newton-Euler Recursion (continued)				
Var.	Waits On	Time at Step End	Step	Cost†
F_i	$g = Avail(\ddot{r}_i)$	$g + SV$	$F_i = m_i\ddot{r}_i$	SV
N_i	$c = Avail(\omega_i)$	$c + MV$	$T_{16} = J_i\omega_i$	MV
	$d = Avail(\dot{\omega}_i)$	$c + MV + VC$	$T_{17} = \omega_i \times T_{16}$	VC
		$d + MV$	$T_{18} = J_i\dot{\omega}_i$	MV
		$\max(c + VC, d) + MV + VA$	$N_i = T_{17} + T_{18}$	VA

Linear Forward Newton-Euler Recursion				
Var.	Waits On	Time at Step End	Step	Cost†
f_i	$h = Avail(f_{i+1})$	$h + MV$	$T_{19} = A_{i+1}f_{i+1}$	MV
		$h + MV + VA$	$f_i = F_i + T_{19}$	VA
n_i	$h = Avail(f_{i+1})$	*already computed* *	$T_{20} = s_i^* \times F_i$	(VC^*)
	$k = Avail(n_{i+1})$	*already computed* *	$T_{21} = T_{20} + N_i$	(VA^*)
		$h + MV + VC$	$T_{22} = p_i^* \times T_{19}$	VC
		$h + MV + VC + VA$	$T_{23} = T_{21} + T_{22}$	VA
		$k + MV$	$T_{24} = A_{i+1}n_{i+1}$	MV
		$\max(k, h + VC + VA) + MV + VA$	$n_i = T_{23} + T_{24}$	VA

* *(that is, computable before the recursion reaches the node, except at the initial node)*

† *VA = time cost of Vector Addition*
VC = time cost of Vector Cross product
SV = time cost of Scalar multiplication of a Vector
MV = time cost of Matrix multiplication of a Vector

Table III.2a — Timing of Linear Newton-Euler Backward Recursion

Timing of ω_i and $\dot{\omega}_i$; $n = 4$, rotational joints

$\omega_0 = \omega_{base}$, $\dot{\omega}_0 = \dot{\omega}_{base}$

(For simplicity here, 1 *Mult* = 1 *Addn* = 1 *Flop*)

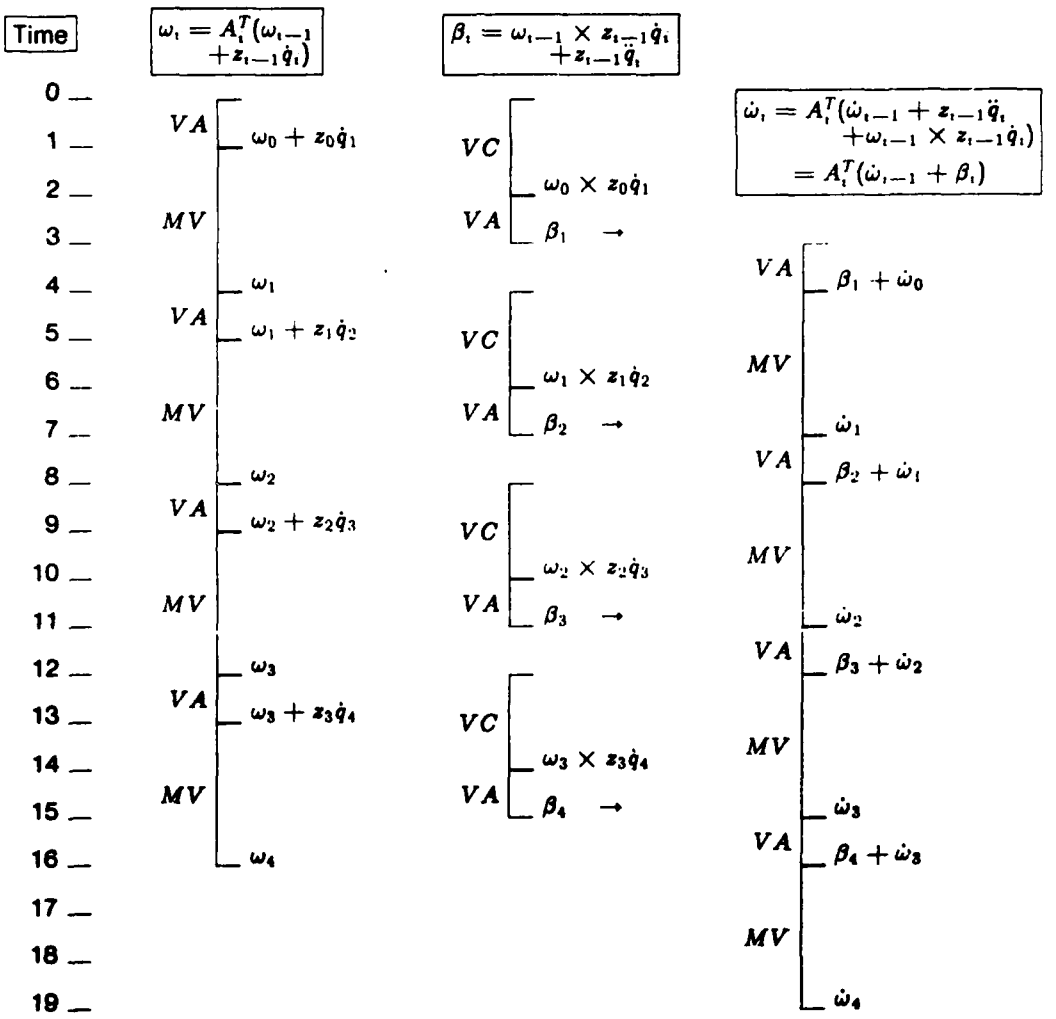


Table III.2a — Timing of Linear Newton-Euler Backward Recursion (continued)
Timing of $\ddot{p}_i$; $n = 4$, rotational joints
$\ddot{p}_0 = \ddot{p}_{base}$

(For simplicity here, 1 *Mult* = 1 *Addn* = 1 *Flop*)

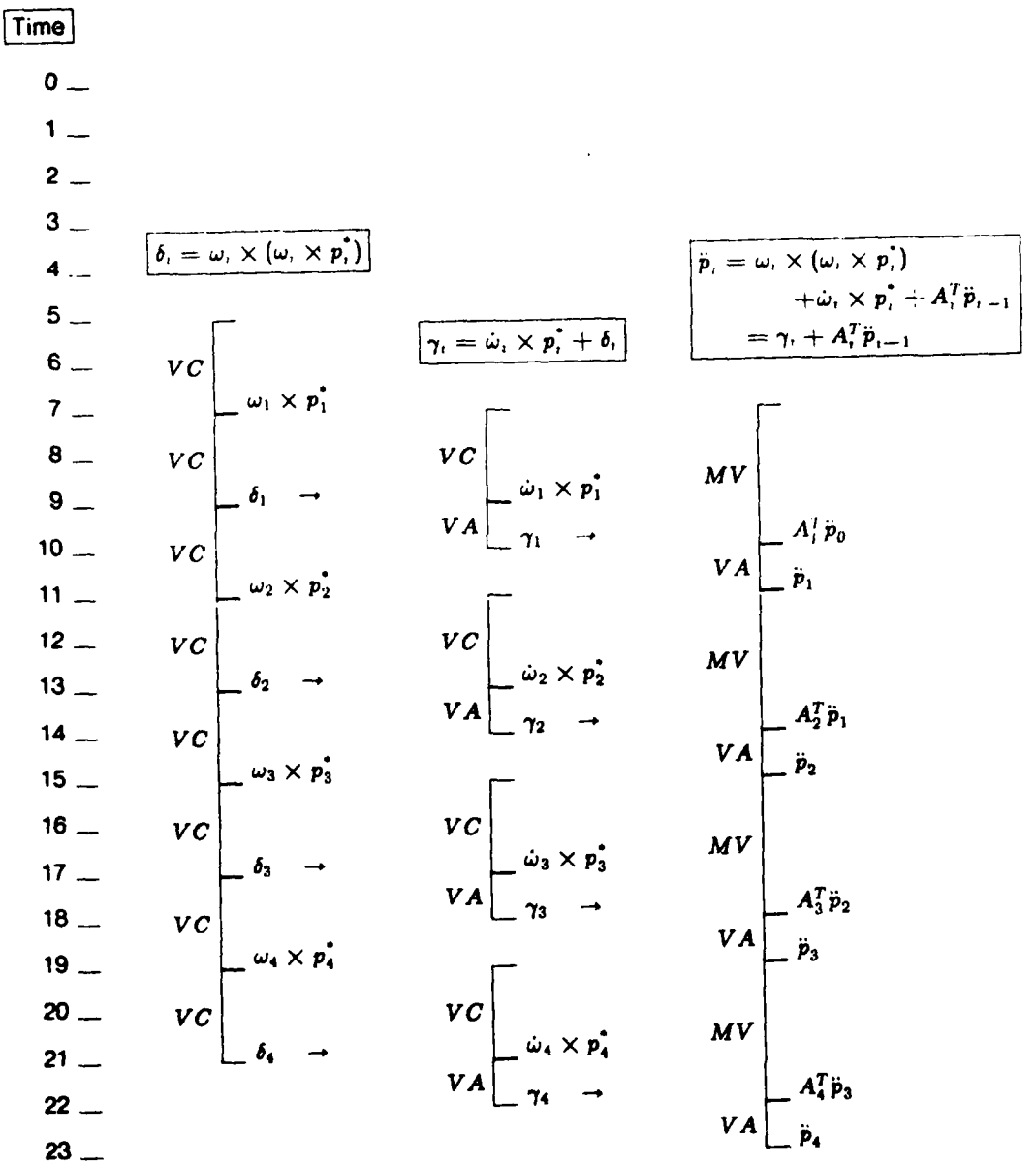


Table III.2a — Timing of Linear Newton-Euler Backward Recursion (continued)
Timing of $\ddot{r}_i$, F_i, N_i; $n = 4$, rotational joints

(For simplicity here, 1 *Mult* = 1 *Addn* = 1 *Flop*)

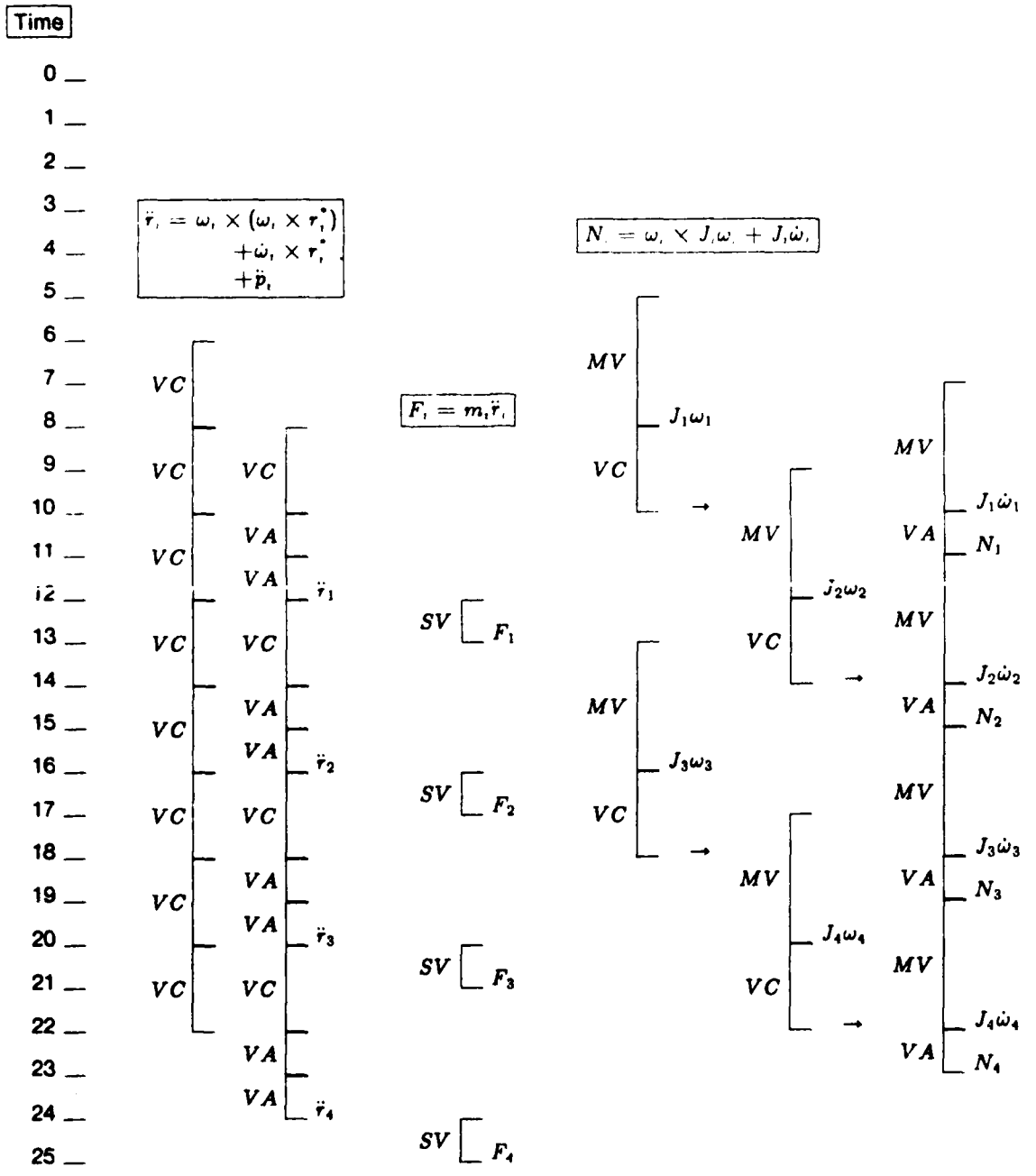
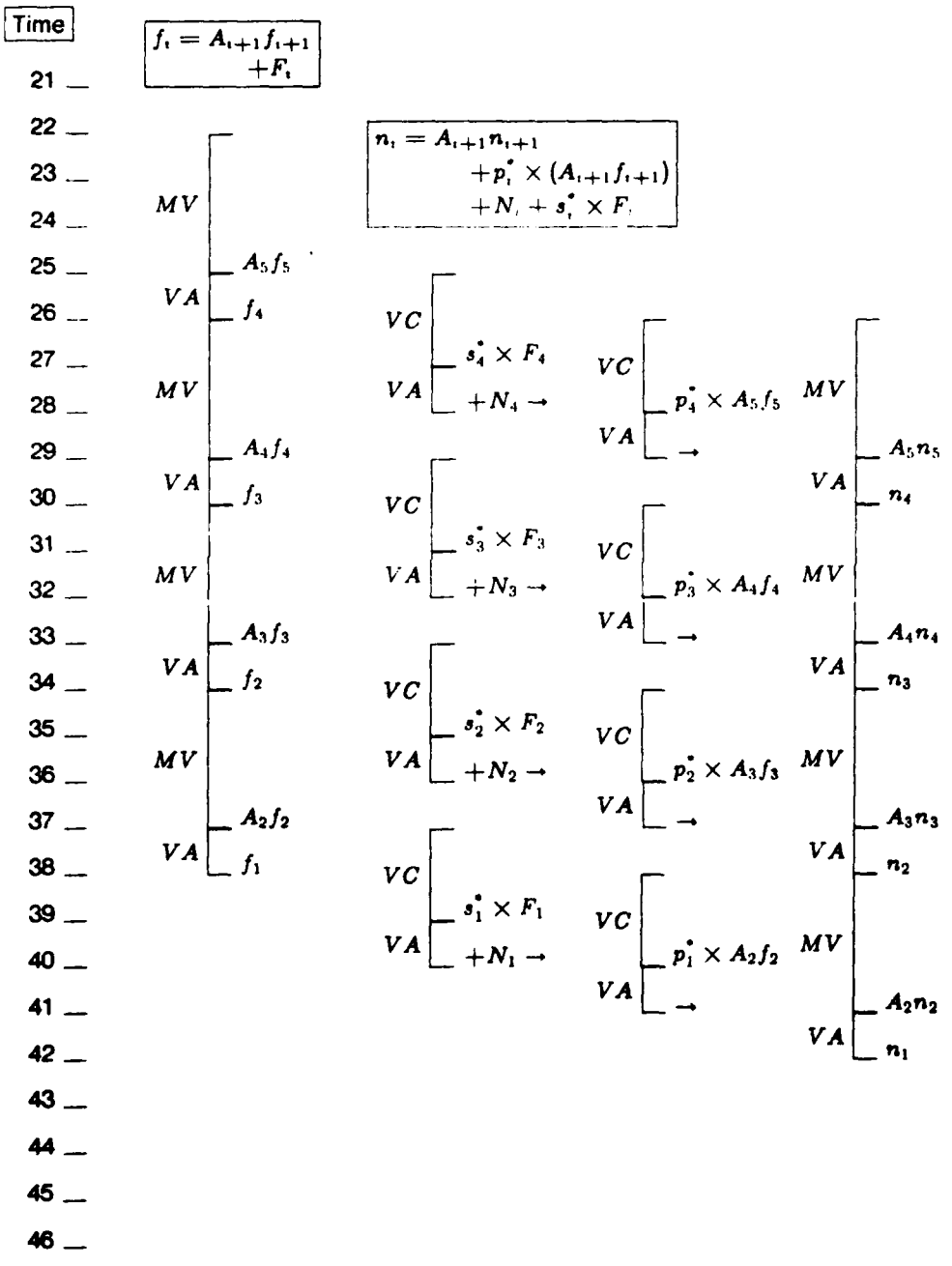


Table III.2b — Timing of Linear Newton-Euler Forward Recursion
Timing of f_i, n_i; $n = 4$, rotational joints
$(f_5 = f_{tip}, n_5 = n_{tip}; \tau_i = [0,0,1]^T \cdot n_i$

(For simplicity here, 1 *Mult* = 1 *Addn* = 1 *Flop*)



("$Bkwd_i$" = backward recursion processor for joint i, "Fwd_i" = forward)

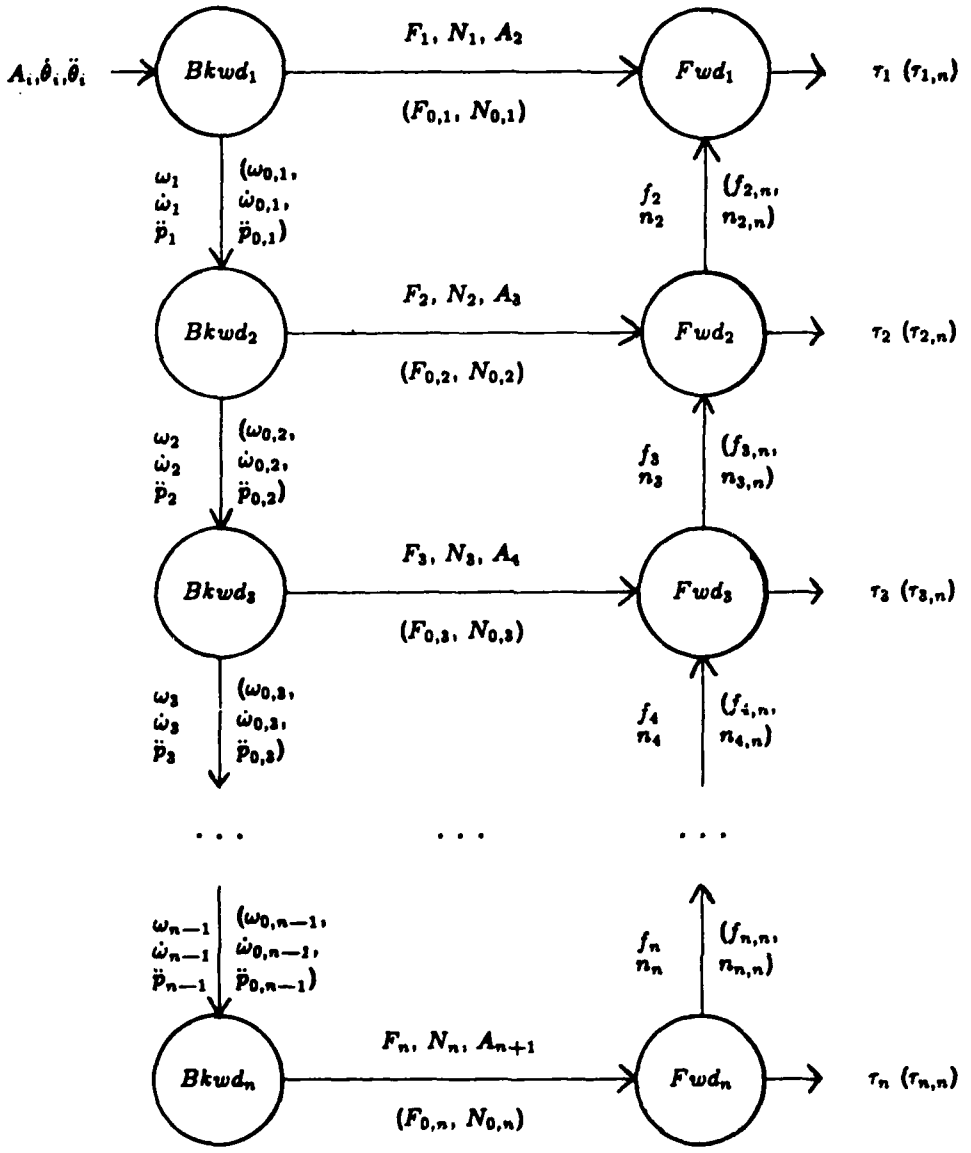
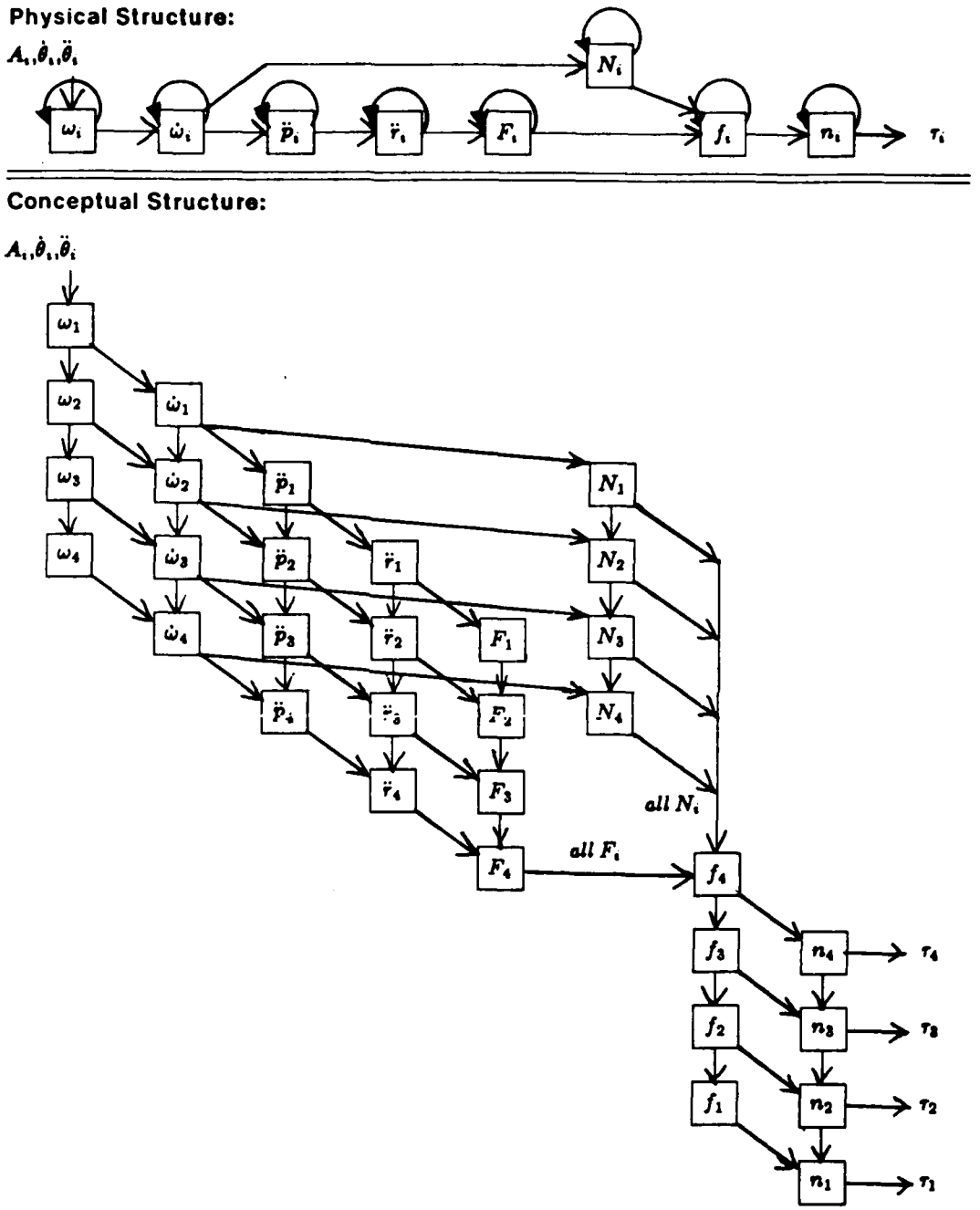
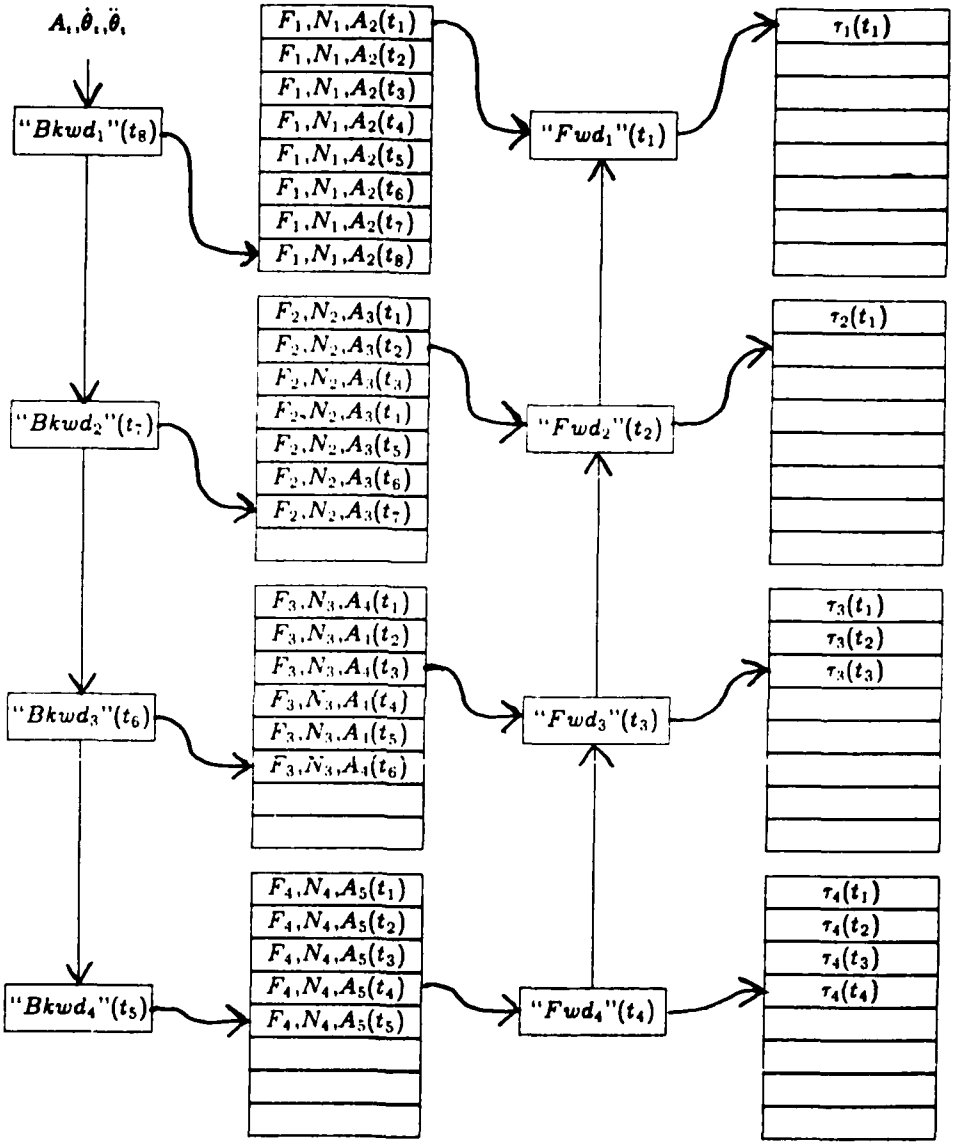


Figure III.1 — Linear Recursive Graph Structure



(Time Intervals $2n(MV + VA) + C$ per complete set of joint torques)
(Only one physical processor per variable — see also Figure VI.1.)

Figure III.2 — Non-Systolic Pipelined Process, $n = 4$



(Time Intervals $1(MV + VA)$ per complete set of joint torques)

("$Bkwd_i$" = backward recursion processor for joint i, "Fwd_i" = forward)

Figure III.3 — Systolic Pipelined Process, $n = 4$

4. PARALLELISM EXPLOITING LOGARITHMIC RECURSION

The preceding section showed that parallelism potentially leads to considerable time savings even within a linear recursive framework. This section will show that the linear time dependency can be improved. By a suitable restructuring of the basic computational framework within which the nodes are embedded, together with a corresponding revision to the recursive forms of the equations, an $\mathcal{O}(\log(n))$ total time may be achieved. For convenient reference, the formulae are collected in Table IV.1.

The basic intuition is illustrated in Figures IV.1.a and b, applied to n consecutive multiplications. If these are performed serially, as in Figure IV.1.a, then time $\mathcal{O}(n)$ is required. By processing in parallel (Figure IV.1.b), time $\mathcal{O}(\log(n))$ may be achieved. It is easy to see that general recursive calculations of the form

$$x_i = a_i x_{i-1} + b_i$$

may be performed in time $\mathcal{O}(\log(n))$.

To exploit this potential in the inverse dynamics case, however, it is necessary to generalize the linear recursive equations. The linear form may be regarded as an operator $\oplus$ which maps a variable (X_{i-1}) representing the base through $(i-1)^{th}$ inputs, together with the i^{th} input (I_i) into (X_i) representing the base through i^{th} inputs:

$$X_{i-1} \oplus I_i \quad \mapsto \quad X_i.$$

For logarithmic recursion, an operator $\otimes$ is required which maps a variable $(X_{a,k})$ representing the a^{th} through k^{th} inputs, together with $(X_{(k+1),b})$ from the $(k+1)^{th}$ through b^{th} inputs, into $(X_{a,b})$ representing inputs a through b:

$$X_{a,k} \otimes X_{(k+1),b} \quad \mapsto \quad X_{a,b}.$$

The linear recursive equation is a special case of the more general logarithmic form in which $a = 0$, $k = i-1$, $b = k+1 = i$, $X_{i,i} = I_i$, and computation proceeds serially, so that

$$\begin{aligned} X_i &= X_{0,i} \\ &= X_{0,(i-1)} \otimes X_{i,i} \\ &= X_{i-1} \oplus I_i. \end{aligned}$$

This is the meaning of the double subscripts in Figure III.1.

What physical meaning can be assigned to this? We will return to this question during the analysis below, where the discussion can be illustrated more clearly by reference to "real" physical quantities. For the present, however, the recursive forms (both linear and logarithmic) may be thought of as mechanisms for relating physical parameters at one coordinate system (or link or joint) to physical parameters at another, by abstracting away the intervening links of the physical manipulator. Very loosely speaking, the goal in both cases is to relate the acceleration of each link to the acceleration of the base by abstracting away the intervening joint accelerations, then to relate the distal forces and torques acting on each link to the distal forces and torques acting on the tip by abstracting away the intervening joint forces and torques. This done, the joint forces and torques necessary to sustain the desired acceleration may be found from a purely local application of Newtonian mechanics. We will use the term "relational parameter" to refer to a quantity which relates physical parameters at one coordinate system to physical parameters at another.

The linear and the logarithmic recursive forms differ principally in how they approach the problem of relating physical parameters between coordinate systems (or links or joints). The linear form relates the base (backward recursion) parameters to the first link parameters to obtain relational parameters of the form X_1 ($\equiv X_{0,1}$). These in turn are related to the second link parameters to obtain relational parameters of the form X_2 ($\equiv X_{0,2}$), which relate the second link to the base. In sequence, the relational parameters $X_{0,3}$, $X_{0,4}$, ..., $X_{0,n}$, are formed, which relate respectively the third, the fourth, and the n^{th} links to the base. The process may be viewed as one which, at each step, "glues" the next link parameter onto the current relational parameters to produce the next relational parameter, thereby relating the base to the next successive link.

In contrast, the logarithmic recursive form may be viewed as a mechanism for "gluing together" *any* two adjacent relational parameters. Parameters of the form $X_{i,i}$ reflect only the input values at link (joint) i; those of the form $X_{i,j}$ reflect (abstract away) links i through j. At the first step, adjacent pairs of (backward recursion) relational parameters of the form $X_{2i,2i}$, $X_{2i+1,2i+1}$ are related in parallel to form the relational parameters $X_{2i,2i+1}$. Thus on the first step we relate alternating pairs of adjacent links to form the relational parameters

$X_{0,1}$, $X_{2,3}$, $X_{4,5}$, ..., $X_{n-1,n}$ (if n odd). At each j^{th} succeeding step, all adjacent pairs of the form $X_{a,k}$, $X_{k+1,b}$ are related in parallel to form the relational parameters $X_{a,b}$, where $a = 2^j m$, $k = a + 2^{j-2}$, $k < b \leq a + 2^j - 1$, and $0 \leq m \leq n/(2^j)$. On the second step, $X_{0,2}$, $X_{0,3}$, $X_{4,6}$, $X_{4,7}$, ..., $X_{n-3,n-1}$, $X_{n-3,n}$ are additionally formed; on the third step we also pick up $X_{0,4}$, $X_{0,5}$, $X_{0,6}$, $X_{0,7}$, $X_{8,12}$, ..., $X_{n-7,n-1}$, $X_{n-7,n}$; and so forth. This process is illustrated in schematic in Figure IV.2 (only the backward recursion is shown). There, $n = 7$, so the backward recursion would take three steps as shown. It is easy to see that in $\lceil \log_2(n+1) \rceil$ steps, all (backward recursion) relational parameters of the form $X_{0,i}$, $0 \leq i \leq n$, may be formed. But as noted above, these are just the linear recursive variables X_i. Thus the backward recursion may be performed in real time in $\mathcal{O}(\log_2(n+1))$ steps using parallel computation. The same is true of the forward recursion, hence of the Inverse Dynamics computation.

The logarithmic recursion operator must possess the following recursive properties:

(a) $X_{a,b}$ must be computable only from inputs a through b,

(b) $X_{a,b} = X_{a,k} \otimes X_{(k+1),b}$ must be computable from variables of the form $Y_{a,k}$ or $Y_{(k+1),b}$ or previously computed $Z_{...}$ or non-recursive values, and

(c) $X_{0,i}$ on the backward recursion, or $X_{i,n}$ on the forward recursion, must be equal to the value of the linear recursive variable X_i.

These properties will allow the use of a structure analogous to Figure IV.2.

The remainder of this section will be devoted to the derivation of appropriate logarithmic recursive formulae. Separate sets of formulae will be developed for the forward and the backward recursion variables. As in Section III, it is necessary to consider only those variables which are propagated the length of the recursive path. For the Newton-Euler formulation these are ω_i, $\dot{\omega}_i$, $\ddot{p}_i$, f_i, and n_i. These formulae are collected in Table IV.1.

Suppose one wished to find the value of non-propagated values, for example to find $\ddot{r}_{i,n}$ for node m on the forward recursion path. One first computes $\omega_{0,m}$, $\dot{\omega}_{0,m}$, and $\ddot{p}_{0,m}$ using the logarithmic forms given in Table IV.1. By the remarks above, these are exactly ω_m, $\dot{\omega}_m$, and

$\ddot{p}_{i,i}$ of Table I.1; and the time needed here to compute all three is $\mathcal{O}(\log(m))$. From these, $\ddot{r}_{n_i}$ may be computed in constant time $\mathcal{O}(c)$ using the formulae in Tables I.1 and III.1.

In the derivations below, for each linear recursive variable, our general strategy will be as follows:

(a) propose a closed-form, non-recursive formula which is equivalent to the linear recursive formula of Table I.1,

(b) show that (a) is correct by showing that it is a fixed point of the linear recursive formula considered, and holds for the zero term (this is equivalent to an inductive proof).

(c) propose a non-linear recursive formula which is equivalent to (a) at $a = 0$ and $b = i$, and

(d) show that the resulting combining operator $\otimes$ is correct by showing that it preserves the form of the formula in (c).

The reader may verify that for $a = 0$, $k = i - 1$, $b = k + 1 = i$ the formulae reduce to Table I.1.

It will be necessary to introduce several auxiliary variables not directly corresponding to any variable in the linear recursive formalism. There we will be concerned to show only that the asserted combining operator is correct.

In the following we assume $a \leq k < b$ throughout, the case of $a = b$ being found as a special case of (c) above. In order to cover both the rotational and translational cases, it is convenient to introduce

$$\sigma_i = \begin{cases} 1 & \text{, joint } i \text{ rotational,} \\ 0 & \text{, joint } i \text{ translational.} \end{cases}$$

$$\bar{\sigma}_i = 1 - \sigma_i$$

NEWTON-EULER BACKWARD RECURSION VARIABLES:

Introduce the auxiliary variable $W_{a,b}$, which will represent products of the coordinate matrices A_i. Notation sometimes used elsewhere in the literature is ${}^{a-1}W_b$, but we write $W_{a,b}$ here for consistency with other variables. Let

$$\begin{aligned} W_{a,b} &\equiv \prod_{j=a}^{b} A_j \\ &= \left(\prod_{j=a}^{k} A_j\right)\left(\prod_{j=(k+1)}^{b} A_j\right) \\ &= W_{a,k}W_{(k+1),b} \end{aligned}$$

It can readily be seen that $W_{i,i}$ maps the coordinates of a vector expressed in the system $\mathcal{O}_i$ into coordinates expressed in $\mathcal{O}_{a-1}$. The physical significance of the combining form $\otimes$ is to compose a mapping which relates $\mathcal{O}_i$ and $\mathcal{O}_k$ with a mapping which relates $\mathcal{O}_k$ and $\mathcal{O}_{a-1}$ in order to produce a mapping which relates the coordinate systems $\mathcal{O}_i$ and $\mathcal{O}_{a-1}$. Thus $W_{a,a} = A_a$ and $W_{a,a-1} = I$ (the identity matrix).

In the case of $W_{a,b}$, the combining operator $\otimes$ is matrix multiplication and

$$W_{a,k} \otimes W_{k+1,b} \equiv W_{a,k}W_{k+1,b}$$

Hereafter we will merely display the combining form without drawing explicit attention to the operator $\otimes$, acknowledging implicitly that $X_{a,b} \equiv X_{a,k} \otimes X_{k+1,b}$.

Next consider the angular velocity ω_i. We define $z_{(-1)}$ to be the axis of rotation of the base system and $\dot{q}_0$ to be its magnitude, so that $\omega_0 = A_0^T z_{(-1)}\dot{q}_0$. This will be non-zero if, for example, it is desired to include the Earth's rotation in the calculations (perhaps for satellite applications). If one creates a fictitious frame $\mathcal{O}_{(-1)}$ at the Earth's center, then $z_{(-1)}$ points through the North Pole and $\dot{q}_0$ is equal to the Earth's angular velocity.

We show that ω_i satisfies the following closed-form non-recursive formula

$$\omega_i = \sum_{j=0}^{i} W_{j,i}^T \sigma_j z_{j-1}\dot{q}_j$$

because this is a fixed-point of the recursive formula for ω_i given in Table I.1.

$$\begin{aligned}\omega_i &= A_i^T\left(\sum_{j=0}^{i-1} W_{j,i-1}^T \sigma_j z_{j-1}\dot{q}_j + \sigma_i z_{i-1}\dot{q}_i\right)\\ &= A_i^T(\omega_{i-1} + \sigma_i z_{i-1}\dot{q}_i)\end{aligned}$$

In order to produce an identical form at $A = 0$ and $b = i$, define

$$\begin{aligned}\omega_{a,b} &\equiv \sum_{j=a}^{b} W_{j,b}^T \sigma_j z_{j-1}\dot{q}_j\\ &= \sum_{j=a}^{k} (W_{j,k}W_{(k+1),b})^T \sigma_j z_{j-1}\dot{q}_j + \sum_{j=k+1}^{b} W_{j,b}\sigma_j z_{j-1}\dot{q}_j\\ &= W_{(k+1),b}^T\omega_{a,k} + \omega_{(k+1),b}\end{aligned}$$

This is the combining form for $\omega_{a,b}$.

$\omega_{a,b}$ expresses the angular velocity of $\mathcal{O}_b$ relative to $\mathcal{O}_{a-1}$, referred to $\mathcal{O}_b$. Thus $\omega_{1,i}$ is the angular velocity of $\mathcal{O}_i$ relative to the base frame, and $\omega_{0,i}$ also accounts for the rotation of the base frame (if $\dot{q}_0 = 0$ then these are the same). The combining form for $\omega_{a,b}$ is a means of composing the angular velocity $\omega_{k+1,b}$ of $\mathcal{O}_b$ relative to $\mathcal{O}_k$ with $\omega_{a,k}$, that of $\mathcal{O}_k$ relative to $\mathcal{O}_{a-1}$, in order to obtain the angular velocity of $\mathcal{O}_b$ relative to $\mathcal{O}_{a-1}$. In particular, $\omega_{i,i-1} = 0$. The rotation matrix $W_{k+1,i}^T$ in the formula transforms $\omega_{a,k}$ from $\mathcal{O}_k$ into $\mathcal{O}_b$, the system to which $\omega_{k+1,b}$ is referred.

Similar remarks apply to the physical significance of $\dot{\omega}_{a,b}$, $\ddot{p}_{a,b}$, etc. Derivations of the other propagated recursion formulae, shown in Table IV.1, are given in Appendix B. Note that ω_0 is the angular velocity of the base, non-zero if the Earth's rotation is modeled as in some satellite applications. Also, $\ddot{p}_0$ is the acceleration of the base. Typically this is the acceleration due to gravity at the site. If one took $\omega_0 \neq 0$ then $\ddot{p}_0$ may also include a term for $\omega_0 \times (\omega_0 \times p_0^*)$, where p_0^* is a vector from the Earth's center $\mathcal{O}_{(-1)}$ to the site; this accounts for the centripetal acceleration arising from the rotation of the Earth. One may account for gravitational acceleration by taking $\ddot{p}_0^g = g$, else $\ddot{p}_i^g = 0$ for $i \neq 0$. The term involving $\ddot{p}_j^g$ is a technical artifice to account for $\ddot{p}_0$ cleanly, and vanishes in the combining form.

As mentioned in Section II on notation, we slightly abuse the dot notation for time differentiation (e.g., $\dot{u}_{a,b}$) to indicate the time differentiation of $u_{a,b}$ from within the coordinate

system $\mathcal{O}_{a-1}$. This is equivalent to the standard notation at $a = 0$ (the case of interest), and results in a less bulky notation. To illustrate this usage, we consider an alternate derivation of $\dot{\omega}_{a,b}$, where $\frac{d^c}{dt}$ denotes time differentiation in $\mathcal{O}_c$, and ${}^c u_{a,b}$ denotes $u_{a,b}$ referred to $\mathcal{O}_c$.

$$\begin{aligned}
\dot{\omega}_{a,b} &= \frac{d^{(a-1)}}{dt}{}^b\omega_{a,b} \\
&= \frac{d^{(a-1)}}{dt}({}^b\omega_{a,k} + {}^b\omega_{k+1,b}) \\
&= {}^b\dot{\omega}_{a,k} + \frac{d^{(a-1)}}{dt}{}^b\omega_{k+1,b} \\
&= W^T_{k+1,b}\dot{\omega}_{a,k} + {}^b\omega_{a,k} \times {}^b\omega_{k+1,b} + \frac{d^k}{dt}{}^b\omega_{k+1,b} \\
&= W^T_{k+1,b}\dot{\omega}_{a,k} + (W^T_{k+1,b}\omega_{a,k}) \times \omega_{k+1,b} + \dot{\omega}_{k+1,b}
\end{aligned}$$

which is our combining form.

NEWTON-EULER FORWARD RECURSION VARIABLES

On the Newton-Euler forward recursion, the coordinate matrix products of interest will be $W_{a+1,k+1}$ instead of $W^T_{k+1,b}$. This is because we wish to transform from $\mathcal{O}_{k+1}$ to $\mathcal{O}_a$, instead of from $\mathcal{O}_k$ to $\mathcal{O}_b$. Intuitively speaking, we are now working from the tip of the manipulator back toward the base. Hence we are desirous of transforming some relational parameter $X_{k+1,b}$, which abstracts the subchain from coordinate system (or link or joint) $k+1$ to b and is expressed in $\mathcal{O}_{k+1}$, into the equivalent quantity expressed in $\mathcal{O}_a$. The $X_{k+1,b}$ so transformed can be combined with $X_{a,k}$, representing the subchain between a and k expressed in $\mathcal{O}_a$, to yield $X_{a,b}$ expressed in $\mathcal{O}_a$. We will here assume that the necessary products $W_{a+1,k+1}$ are generated on the forward recursion in accord with the formulae above. This requires a minor abuse of notation, as the combining form would then be

$$W_{a+1,b+1} = W_{a+1,k+1} \otimes W_{k+2,b+1}$$

but we have agreed that our combining forms will be

$$X_{a,b} = X_{a,k} \otimes X_{k+1,b}.$$

If the reader finds this troublesome, we suggest that she or he make the substitutions $a' = a + 1$, $b' = b + 1$, and $k' = k + 1$. This done, one need only remember that

$$W_{a',a'} = A_{a+1}.$$

Table IV.1 — Logarithmic Recursive Formulations

Logarithmic Backward Recursion:

$$\sigma_a = \begin{cases} 1 & \text{, joint } a \text{ rotational,} \\ 0 & \text{, joint } a \text{ translational.} \end{cases}$$

$$\bar{\sigma}_a = 1 - \sigma_a$$

$$W_{a,b} = \begin{cases} A_a & \text{if } a = b; \\ W_{a,k} W_{(k+1),b} & \text{if } a \neq b. \end{cases}$$

$$\omega_{a,b} = \begin{cases} \sigma_a A_a^I z_{(a-1)} \dot{q}_a & \text{if } a = b; \\ W_{(k+1),b}^T \omega_{a,k} + \omega_{(k+1),b} & \text{if } a \neq b. \end{cases}$$

$$\dot{\omega}_{a,b} = \begin{cases} \sigma_a A_a^I z_{(a-1)} \ddot{q}_a & \text{if } a = b; \\ W_{(k+1),b}^T \dot{\omega}_{a,k} + (W_{(k+1),b}^T \omega_{a,k}) \times \omega_{(k+1),b} + \dot{\omega}_{(k+1),b} & \text{if } a \neq b. \end{cases}$$

$$Q_{a,b} = \begin{cases} \bar{\sigma}_a A_a^T z_{a-1} \dot{q}_a & \text{if } a = b; \\ W_{(k+1),b}^T Q_{a,k} + Q_{(k+1),b} & \text{if } a \neq b. \end{cases}$$

Table IV.1 — Logarithmic Recursive Formulations (continued)

Logarithmic Backward Recursion (continued):

$$R_{a,b} = \begin{cases} p_a^* & \text{if } a = b; \\ W_{(k+1),b}^T R_{a,k} + R_{(k+1),b} & \text{if } a \neq b. \end{cases}$$

$$S_{a,b} = \begin{cases} \omega_{a,a} \times p_a^* & \text{if } a = b; \\ W_{(k+1),b}^T S_{a,k} + (W_{(k+1),b}^T \omega_{a,k}) \times R_{(k+1),b} + S_{(k+1),b} & \text{if } a \neq b. \end{cases}$$

$$\ddot{p}_{a,b} = \begin{cases} \ddot{p}_a^J + \dot{\omega}_{a,a} \times p_a^* + \omega_{a,a} \times (\omega_{a,a} \times p_a^*) + \bar{\sigma}_a A_a^T z_{a-1} \ddot{q}_a + 2\omega_{a,a} \times Q_{a,a} & \text{if } a = b; \\ W_{(k+1),b}^J \ddot{p}_{a,k} + \ddot{p}_{(k+1),b} + (W_{(k+1),b}^J \dot{\omega}_{a,k}) \times R_{(k+1),b} & \\ \quad + (W_{(k+1),b}^T \omega_{a,k}) \times \left((W_{(k+1),b}^T \omega_{a,k}) \times R_{(k+1),b} + 2(S_{(k+1),b} + Q_{(k+1),b}) \right) & \text{if } a \neq b \end{cases}$$

Logarithmic Forward Recursion:

$$f_{a,b} = \begin{cases} F_a & \text{if } a = b; \\ W_{a+1,k+1} f_{(k+1),b} + f_{a,k} & \text{if } a \neq b. \end{cases}$$

$$n_{a,b} = \begin{cases} N_a + s_a^* \times F_a & \text{if } a = b; \\ n_{a,k} + W_{a+1,k+1} \left(n_{(k+1),b} + (A_{k+1}^T R_{a,k}) \times f_{(k+1),b} \right) & \text{if } a \neq b. \end{cases}$$

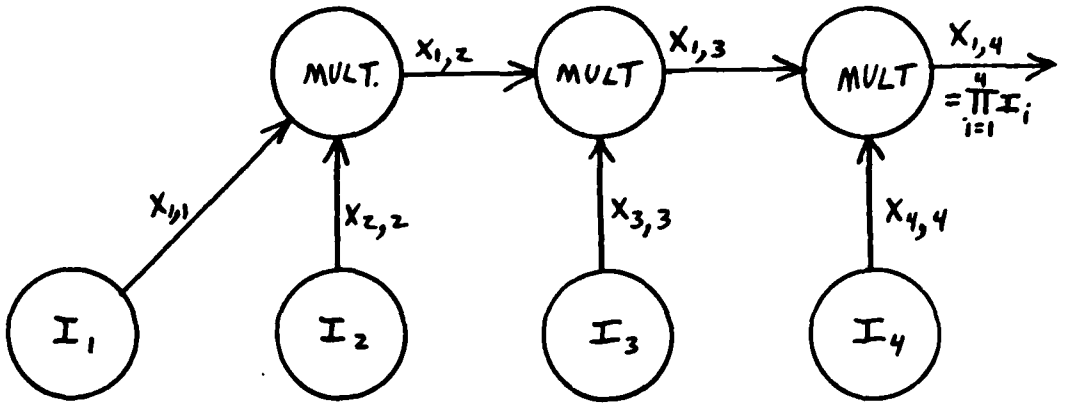


Figure IV.1a — Serial Vs. Parallel Multiplication (Serial)

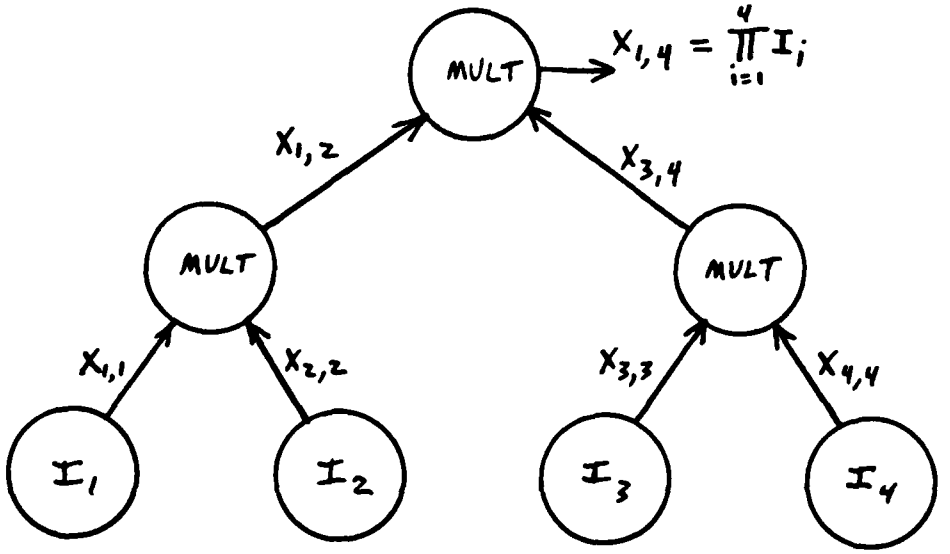


Figure IV.1b — Serial Vs. Parallel Multiplication (Parallel)

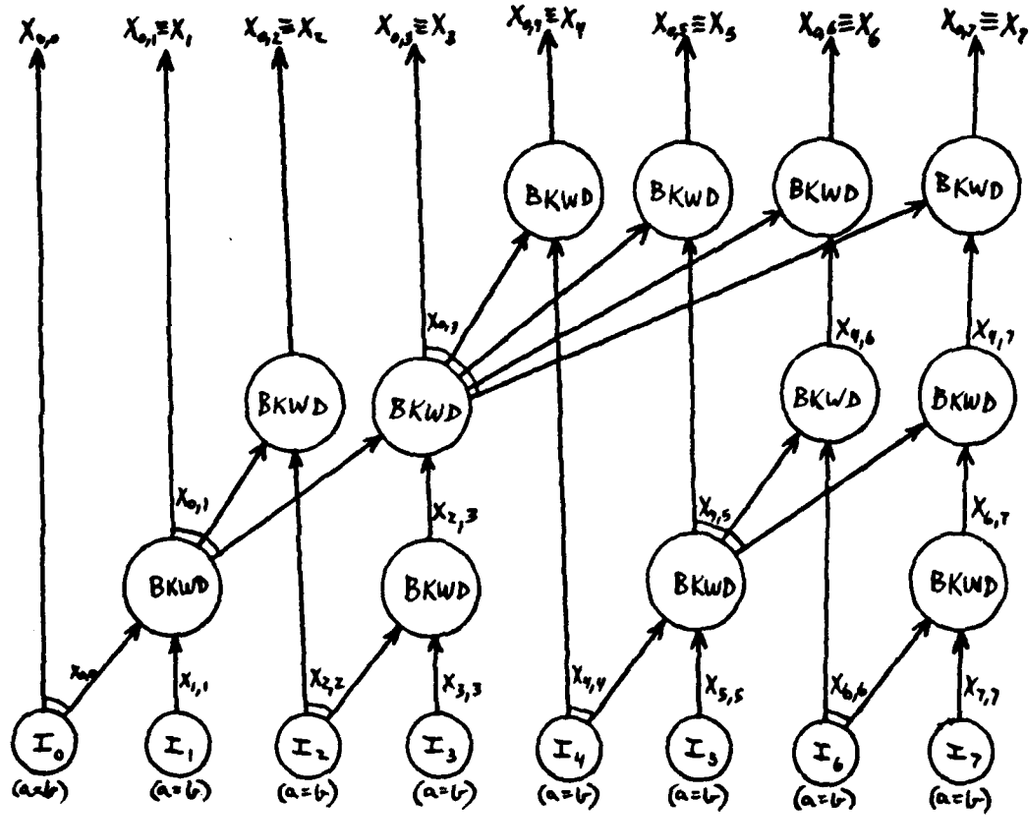


Figure IV.2 — Logarithmic Recursive Graph Structure

(Only backward recursion shown)

Because here the extension to the translational case may not be entirely obvious, the equations and analyses presented cover both rotational and translational joints in manipulators composed of mixed systems. Analogously to Section III, as a conceptual aid we assume that there is one group of parallel processors for each node shown in Figure IV.2, and also one group for each node in the similar forward recursion. However, only one row (tier) is active in the computation at any one step, and all tiers are identical. Thus an implementation could be constructed using only one physical device for each pair of joints of the manipulator, by connecting the outputs back to the inputs through a buffer. Implementation details are expanded further in Section VI. The comments about the 4 *Flop* systolic pipeline interval, found in the introduction to Section III, also apply here.

The detailed internal structure of a node is presented in Table V.1, which is analogous to Table III.1. Note that the formulae presented in Table IV.1 for the forward recursion require calculating $W_{i+1,i+1}$ and $R_{i,i}$ again, exactly as was done on the backward recursion. These have no interesting data dependencies, do not bound the computation, and are computed exactly the same on the forward as on the backward recursion — thus for conciseness they are shown only once in Table V.1.

Delay conditions for the logarithmic case proceed as shown in Appendix C, similarly to Section III.

Since it is possible to satisfy the minimum delay conditions, propagation occurs at a rate of $(MV + VA)$ per node. Assuming that the delay conditions are satisfied and that all input becomes available simultaneously, it can readily be seen that

$$Avail(X_{a,b}) = Avail(X_{a,a}) + \lceil \log_2(b - a + 1) \rceil (MV + VA).$$

Thus in particular, if X_i is the linear recursive variable corresponding to $X_{a,b}$, then $X_i = X_{0,i}$ so

$$\begin{aligned} Avail(X_i) &= Avail(X_{0,i}) \\ &= Avail(X_{a,a}) + \lceil \log_2(i + 1) \rceil (MV + VA). \end{aligned}$$

Assuming a maximally parallel implementation as before, the total time of the calculations

is shown in Appendix C to be

$$2\lceil \log_2(n+1) \rceil (1\ Mult + 3\ Addns) + 5\ Mults + 10\ Addns.$$

For the same reasons as in Section III this is slightly lower (by $2\ Mults - 2\ Addns$) than given in [25], due to differences (not requiring additional computation by the host) in assumptions about the form of the input and output (not in the computation).

Table V.1 — Relative Time of Logarithmic Data Dependencies

Logarithmic Recursive Forms; $a \neq b$

Logarithmic Backward Newton-Euler Recursion				
Var.	Waits On‡	Time at Step End	Step	Cost†
$W_{a,b}$	$a = Avail(W_{x,y})$	$a + MM$	$W_{a,b} = W_{a,k} W_{k+1,b}$	MM
$\omega_{a,b}$	$a = Avail(W_{x,y})$	$\max(a,b) + MV$	$T_1 = W^T_{k+1,b} \omega_{a,k}$	MV
	$b = Avail(\omega_{x,y})$	$\max(a,b) + MV + VA$	$\omega_{a,b} = T_1 + \omega_{k+1,b}$	VA
$\dot{\omega}_{a,b}$	$a = Avail(W_{x,y})$	$\max(a,c) + MV$	$T_2 = W^T_{k+1,b} \dot{\omega}_{a,k}$	MV
	$b = Avail(\omega_{x,y})$	$\max(a,b) + MV + VC$	$T_3 = T_1 \times \omega_{k+1,b}$	VC
	$c = Avail(\dot{\omega}_{x,y})$	$\max(a + MV + VC,$ $b + MV + VC, c) + VA$	$T_4 = T_3 + \dot{\omega}_{k+1,b}$	VA
		$\max(a + VC + VA,$ $b + VC + VA,$ $c) + MV + VA$	$\dot{\omega}_{a,b} = T_2 + T_4$	VA
$R_{a,b}$	$a = Avail(W_{x,y})$	$\max(a,d) + MV$	$T_5 = W^T_{k+1,b} R_{a,k}$	MV
	$d = Avail(R_{x,y})$	$\max(a,d) + MV + VA$	$R_{a,b} = T_5 + R_{k+1,b}$	VA
$S_{a,b}$	$a = Avail(W_{x,y})$	$\max(a + MV, b + MV,$ $d) + VC$	$T_6 = T_1 \times R_{k+1,b}$	VC
	$b = Avail(\omega_{x,y})$			
	$d = Avail(R_{x,y})$	$\max(a,e) + MV$	$T_7 = W^T_{k+1,b} S_{a,k}$	MV
	$e = Avail(S_{x,y})$	$\max(a + MV + VC,$ $b + MV + VC,$ $d + VC, e) + VA$	$T_8 = T_6 + S_{k+1,b}$	VA
		$\max(a + MV + VC + VA,$ $b + MV + VC + VA,$ $d + VC + VA,$ $e + MV) + VA$	$S_{a,b} = T_7 + T_8$	VA
$Q_{a,b}$	$a = Avail(W_{x,y})$	$\max(a,f) + MV$	$T_9 = W^T_{k+1,b} Q_{a,k}$	MV
	$f = Avail(Q_{x,y})$	$\max(a,f) + MV + VA$	$Q_{a,b} = T_9 + Q_{k+1,b}$	VA

† See end of continued table.

‡ $X_{x,y}$ = both of $X_{a,k}$ and $X_{k+1,b}$

Table V.1 — Relative Time of Logarithmic Data Dependencies (continued)

Logarithmic Recursive Forms; $a \neq b$

Logarithmic Backward Newton-Euler Recursion				
Var.	Waits On‡	Time at Step End	Step	Cost†
$\ddot{p}_{x,y}$	$a = Avail(V_{x,y})$	$\max(a + MV, c + MV, d) + VC$	$T_{10} = T_2 \times R_{k+1,b}$	VC
	$b = Avail(\omega_{x,y})$	$\max(e, f) + VA$	$T_{11} = S_{k+1,b} + Q_{k+1,b}$	VA
	$c = Avail(\dot{\omega}_{x,y})$	$\max(e, f) + VA + SV$	$T_{12} = 2T_{11}$	SV
	$d = Avail(R_{x,y})$	$\max(a + MV + VC, b + MV + VC,$	$T_{13} = T_6 + T_{12}$	VA
	$e = Avail(S_{x,y})$	$d + VC, e + VA + SV,$		
	$f = Avail(Q_{x,y})$	$f + VA + SV) + VA$		
	$g = Avail(\ddot{p}_{x,y})$	$\max(a + MV + VC, b + MV + VC,$	$T_{14} = T_1 \times T_{13}$	VC
		$d + VC, e + VA + SV,$		
		$f + VA + SV) + VC + VA$		
		$\max(a + MV + 2VC + VA,$	$T_{15} = T_{10} + T_{14}$	VA
		$b + MV + 2VC + VA,$		
		$c + MV + VC, d + 2VC + VA,$		
		$e + VC + 2VA + SV,$		
		$f + VC + 2VA + SV) + VA$		
		$\max(a, g) + MV$	$T_{16} = W^I_{k+1,b}\ddot{p}_{...}$	MV
		$\max(a + MV + 2VC + 2VA,$	$T_{17} = \ddot{p}_{k+1,b} + T_{15}$	VA
		$b + MV + 2VC + 2VA,$		
		$c + MV + VC + VA, d + 2VC + 2VA,$		
		$e + VC + 3VA + SV,$		
		$f + VC + 3VA + SV, g) + VA$		
		$\max(a + MV + 2VC + 3VA,$	$\ddot{p}_{a,b} = T_{16} + T_{17}$	VA
		$b + MV + 2VC + 3VA,$		
		$c + MV + VC + 2VA, d + 2VC + 3VA,$		
		$e + VC + 4VA + SV,$		
		$f + VC + 4VA + SV, g + MV) + VA$		

† See end of continued table.

‡ $X_{x,y}$ = both of $X_{a,k}$ and $X_{k+1,b}$

Table V.1 — Relative Time of Logarithmic Data Dependencies (continued)

Logarithmic Recursive Forms; $a \neq b$

Logarithmic Forward Newton-Euler Recursion				
Var.	Waits On‡	Time at Step End	Step	Cost†
$f_{a,b}$	$h = Avail(f_{i,j})$	$h + MV$	$T_{19} = W_{a+1,k+1} f_{k+1,b}$	MV
		$h + MV + VA$	$f_{a,b} = f_{a,k} + T_{19}$	VA
$n_{i,j}$	$h = Avail(f_{i,j})$	*(already computed *)*	$T_{20} = A^I_{k+1} R_{a,k}$	(MV^*)
	$i = Avail(n_{\ldots})$	$h + VC$	$T_{21} = T_{20} \times f_{k+1,b}$	VC
		$h + MV + VC$	$T_{22} = W_{i+1,k+1} T_{21}$	MV
		$\max(i, h + MV + VC) + VA$	$T_{23} = n_{a,k} + T_{22}$	VA
		$i + MV$	$T_{24} = W_{i+1,k+1} n_{k+1,b}$	MV
		$\max(i, h + VC + VA) + MV + VA$	$n_{a,b} = T_{23} + T_{24}$	VA

* *(that is, computable before the recursion reaches the node, except at the initial node)*

‡ $X_{i,j}$ = both of $X_{a,k}$ and $X_{k+1,b}$

† VA = *time cost of Vector Addition*
VC = *time cost of Vector Cross product*
SV = *time cost of Scalar multiplication of a Vector*
MV = *time cost of Matrix multiplication of a Vector*
MM = *time cost of Matrix Multiplication*

6. IMPLEMENTATION CONSIDERATIONS

As noted earlier, the main thrust of this paper has been an exploration of the extent to which parallelism can be pushed in this particular domain. Nonetheless, we hope to indicate that the physical requirements are not particularly excessive and thus this might be a reasonable thing to actually implement. This section contains a fairly general discussion of hardware requirements. Chip architecture is discussed in the next section.

Except perhaps for some variation in the constant term, the hardware sketched in this section is intended to capture the maximally parallel nature of the algorithms and to physically attain the stated time bounds. It is always possible to deliberately sacrifice speed and gain material economy by allowing hardware multiplexing in a partially parallel system.

The description of the algorithms, the data dependencies, and the timing, have been developed quite generally in terms of matrix and vector operations. One can easily imagine many different ways of implementing matrix arithmetic, however. The purpose of this section will be to imagine these in sufficient detail to conclude that the algorithms proposed are reasonable. We also imagine that any actual physical implementation will differ in many particular details from those presented here. That is, we expect that the details of the arrangements and requirements which we display will change, but not by so much as to render the general conclusions inapplicable.

Much that we will observe, however, can be seen as a consequence of the macroscopic computational structure of the algorithm itself. Where possible we will conduct the discussion at a level of matrix-vector arithmetic whose operators are expressed in differing implementations, and we will seek to discover regularities which constrain the design across different implementations.

We will generally worry about three main concerns which commonly dog parallel systems:

(a) internal buffering and storage of intermediate results,

(b) communication and bussing of intermediate results, and

(c) number of processors.

We will propose a general computational structure, then use that to consider requirements

for internal buffering and storage. This will lead us to the communication and bussing necessary to get the data to the appropriate storage locations. Finally we consider the processing required.

We accept several intuitions commonly held in VLSI, including: as dimensions scale downward complexity scales upward, which leads us to seek simplicity and regularity; and as dimensions scale downward processing power becomes cheap and communication, bussing, and buffering become the limiting factors. We are for now willing to assume global communication (in particular, a global clock and a global reset signal). This avoids many problems of timing and handshaking protocol. We further assume that timewise, 1 $Mult$ = 1 $Addn$ = 1 $Flop$. This substantially simplifies timing, and would not be an unreasonable simplification to impose on a (synchronous) physical implementation anyway. In the discussion of buffering in particular, we will use this metric (together with the relative offsets and more general timing developed in Sections III and V) to establish the cases in which buffering is or is not required.

Note that the logarithmic formalism as developed in the text above required a separate handling of the $a = b$ case. This would certainly be the way one would build it from multipliers and adders in order to achieve the tightest possible time bound. However, if the most likely implementation of the logarithmic algorithm will be in VLSI or WSI it is here more interesting to devise a single regular, systematic structure which uniformly handles both the $a = b$ and the $a \neq b$ cases. Since $a = b$ only happens once each direction we are quite happy to purchase simplicity and regularity at the price of an increase in the constant term. Appendix D presents a technical artifice by which we can make the $a = b$ case look like $a \neq b$.

The conceptual structure of the computation was shown in Figure III.1 (for the linear case) and Figure IV.2 (for the logarithmic case, with only the backward recursion shown). The processor nodes could be hooked up in a regular, regularly extensible structure very similar to Figures III.1 and IV.2. This would permit global pipelining of successive sets of joint torques at intervals of 4 $Flops$. However, note that only one node (linear) or tier (logarithmic) is ever active at any given point in the computation. We could therefore map our full conceptual structure into a much smaller one, which buffers intermediate results in such a

way as to use only one physical processor node (linear) or tier of nodes (logarithmic). This would use much less hardware, but would imply that computation of one set of joint torques must complete before the next could begin. The basic notion (which will be expanded more fully in the discussion below) is shown in Figure VI.1a, for conceptual orientation. The backward and forward recursion nodes are never simultaneously active, so processors could be shared between them; see Figure VI.1b. The choice of architecture will depend heavily on whether it is desired to systolically pipeline successive sets of joints torques at 4 *Flop* intervals, or simply to calculate one set.

BUFFERING

As shown in Figure VI.1b, there are four sets of buffers and communication pathways with which we will concern ourselves: input buffering, intra-node buffering, inter-node buffering, and backward-forward recursion buffering. These may be roughly divided into internal and external buffers. From input/output considerations we can put immediate bounds on the amount of buffering external to a node which may be required (this is input, inter-node, and backward-forward recursion buffering). There are, after all, only so many variables to transmit. The potential problem lies in the internal buffering within any one processor node, for we have created a whole host of intermediate partial computations which have complicated, intertwined data dependencies. The potentially massive buffering required for these is actually very tightly bounded.

Input buffering requires storing the scalar values for ${}^{i-1}\dot{q}_i$, ${}^{i-1}\ddot{q}_i$, and A_i, as well as the manipulator configuration parameters (p_i^*, $\ddot{p}_i^g$, r_i^*, s_i^*, z_{i-1}, m_i, and J_i) and the endpoint values (ω_0, $\dot{\omega}_0$, $\ddot{p}_0$, f_{n+1} and n_{n+1}), for a total of about $37n + 15$ scalar values. Buffering within any one node in the linear case requires at most 15 scalar values of temporary storage since (referring to Table III.1) only ω_i, T_8, T_{13}, T_{15} and T_{17} are not used immediately, but this requirement can be met by simply latching the output of the sub-processors into registers (as in the next section). This requirement is only 3 scalars in the logarithmic case (for T_1), and it can also be met by latching. Buffering between nodes is not needed in the linear case, and requires about 54 scalars per node in the logarithmic case (if outputs are latched, this can nearly be halved). Backward-forward recursion buffering involves at most only the

passed recursion variables from each joint. (Additional intermediate storage is needed if a systolic pipelined architecture is implemented, by about a factor of n, linear, or $\lceil \log_2 n \rceil$, logarithmic.) Buffering requirements are considered in more detail in [25].

COMMUNICATION

We do not wish to propose a detailed communication protocol, only to show that the requirements are sufficiently bounded that such a protocol could be devised. To develop this, we shall discuss communication in terms of "wires" which are vector busses three scalars wide, and not allow multiplexing of busses. In a real implementation, data on the "vector busses three scalars wide" might be transmitted serially as three sequential numbers over only a single real wire. Also in a real implementation, the busses might be time multiplexed. Thus in the discussion below, the reader should supply her or his own scale factor depending on her or his own implementation image. After developing communication models for Figures VI.1b, we will show how the systolic pipelined structures of Figures III.1 and IV.2 have a regular, regularly extensible communication network.

As in the question of buffering, communication falls into that internal and external to a node. The internal communications may be treated as comprising a fixed-size hard-wired module in which the "wires" are laid down as dictated by the algorithm. The external communication must carry externally buffered data back and forth between the process nodes and the buffers.

Bounds may be placed on the internal communication by noting the following:

> (a) as shown below (when number of processors is considered), none of the linear forward or backward recursion, the logarithmic forward or backward recursion, or the non-propagated, variables have more than 25 matrix-vector operators,
>
> (b) no operator has more than two operands, and therefore
>
> (c) no single module of the above (a) has more than 50 point-to-point busses.

If the forward and backward recursion share processors then this must be increased to a

maximum of 35 operators, hence 70 busses. The non-propagated variables have no more than 10 operators, hence 20 busses, and these must be included somewhere as noted above. We stress that this is an upper bound based only on the total number of processors, and in any real implementation the actual number would doubtless be much smaller. This is because many intermediate results are used by only one operator, so we expect that many operands could be transmitted by abutment, by simply connecting one operator's output directly to the input of the next operator. This will, however, depend on the particular implementation layout geometry chosen. The essential point is that there are easily few enough wires to route on a point-to-point basis.

For external communication we consider the paths to and from the buffers separately. We essentially argue that since the external buffering requirements were not excessive, the communication required to support those requirements will not be excessive. No inter-node buffering was required in the linear case, but the variables themselves must be transmitted — this requires four vectors on the backward recursion and two on the forward. The logarithmic case required buffering at most 18 vectors (54 scalars) per node per $(MV + VA)$ cycle (both backward and forward recursion), hence at most 18 vector busses. The input values (18 scalars per node) and the endpoint values (ω_0, etc. — 12 scalars at the first node of the backward recursion and 6 scalars at the first node of the forward) must be communicated. Finally, each node must eventually communicate F_i and N_i to the backward-forward recursion buffers.

The systolic pipelined structure of Figures III.1 and IV.2 has particularly nice scaling properties as n increases. These structures do not fold the nodes or tiers together as does Figure VI.1a or b. In the logarithmic case, the reader should compare Figure IV.2 with Figures VI.2 and VI.3. All show almost identical structure, but in different fashion. Figure VI.2 shows Figure IV.2 expanded to include the forward recursion, and Figure VI.3 maps this into a regular rectilinear array. This array is also regularly extensible as shown in Figure IV.4. In all cases, circles are processor nodes which implement one complete node of the logarithmic algorithm. Additionally, in Figure VI.3 squares represent passive buffers which do no more than perform buffering for the variables transmitted from node to node, or from backward to forward recursion. In the linear case the regular, regularly extensible array

structure can be clearly seen from Figure III.1. Passive buffering is only required between backward and forward recursion processors. (We assume that the non-propagated variables are computed from the propagated variables within the circles as appropriate, in the manner discussed above). All variables pertaining to the same time slice are thus calculated in the same node (linear) or tier (logarithmic), and progress node by node (tier by tier) until they finally emerge as the desired torques. Each node or tier requires $(MV + VA) = 4$ *Flops* to complete, so if different successive input values were presented at intervals of 4 *Flops* at the bottom, corresponding motor torques would emerge from the top at intervals of 4 *Flops*. It seems likely that the speed would be bounded by the input/output requirements of the host system.

The thing to notice about the linear case in Figure III.1 is that it is possible to restrict the total width of the busses between successive nodes to the width required for the inter-node communication of any single node, and as argued above this is finite and small. In the logarithmic case (Figure VI.3), the maximum total width of the busses may be restricted to twice this. Thus as the structure is scaled upward (i.e. as n becomes arbitrarily large), the amount of area consumed by busses remains a constant fraction of the total area.

NUMBER OF PROCESSORS

Consider next the number of processors required. Tables III.1 and V.1 were constructed to represent exactly each operation at a node exactly once (except that as noted in Section V. $W_{i,v}$ and $R_{i,v}$ are calculated in the same way on both the logarithmic backward and forward recursions, so for conciseness were shown only on the backward — here they must be counted in the forward recursion).

Since the time denotation given in Tables III.1 and V.1 (MV, VA, VC, etc.) also denotes the operation, we see that to propagate the recursive variables through a single node requires

Requirements Per Node — Propagated Variables Only						
Algorithm	MV	VA	VC	SV	MM	Total
Linear Bkwd. N.E.	3	5	4			12
Linear Fwd. N.E.	2	4	2			8
Logarithmic Bkwd. N.E.	6	12	4	1	1	24
Logarithmic Fwd. N.E.	5	4	1		1	11

Additionally, one must also compute the non-propagated variables ($\ddot{r}_i$, F_i, N_i, and τ_i from Table III.1), which is the same for both the linear and the logarithmic cases.

Requirements Per Node — Non-Propagated Variables Only						
Algorithm	MV	VA	VC	SV	MM	Total
Backward Recursion	2	3	4	1		10
Forward Recursion						0

The total per-node requirements are thus

Requirements Per Node — Totals						
Algorithm	MV	VA	VC	SV	MM	Total
Linear Bkwd. N.E.	5	8	8	1		22
Linear Fwd. N.E.	2	4	2			8
Logarithmic Bkwd. N.E.	8	15	8	2	1	34
Logarithmic Fwd. N.E.	5	4	1		1	11

An interesting optimization is possible in the logarithmic case if it is not required to model the rotation of the earth and if n is even. Recall from Section VI that the recursion was grounded in $O_{(-1)}$, and that we considered $z_{(-1)}$ as a vector pointing through the North Pole, etc. This was primarily done to provide a firm and unambiguous grounding for the mathematical analysis, and in many cases may not be required. The logarithmic algorithm

requires one processor node for each pair (or fraction thereof) of input nodes. If n is odd then $\lceil (n/2) \rceil = \lceil (n+1)/2 \rceil$ and the number of processor nodes is $(n+1)/2$ regardless. However, if n is even then $\lceil (n/2) \rceil = \lceil (n+1)/2 \rceil - 1$, and by grounding the recursion in $\mathcal{O}_0$ we may eliminate one processor node. Except perhaps in satellite applications, typically this loses nothing interesting anyway.

An implementation might involve a special-purpose VLSI chip capable of handling general vector arithmetic up to and including matrix-vector multiplication. (This is chosen to be midway between two alternatives — a matrix-matrix chip would be larger and could be implemented using three of the proposed chips, while a vector-vector chip would be smaller but three could implement the chip proposed.) The chip would incorporate 18 registers (for storing three pairs of 3-vectors), 6 multipliers and 3 adders. It would also need a means of sequencing these operations, as well as a means of decoding (perhaps from jumpers wired to the pins) which particular vector operation to perform.

A natural candidate is a datapath chip, as in Barrett et al.[3]. As shown in Figure VI.5, it consists of a Programmable Logic Array (PLA) which decodes micro-instructions to produce control signals driving computational and storage elements attached to several common data busses. By controlling when the storage elements read and write which busses, and when and from which busses the computational elements obtain their operands and output their results, the nature of the computation performed on-chip is controlled. Thus a datapath chip functions as an easily-customizable micro-CPU. This will be explored more fully in the next section.

The reader can gain some intuitive feel for an implementation by turning to Table III.2, and imagining that the pages represent printed-circuit boards and the indicated operations represent chips. The structure so composed would implement the fully systolic pipelined linear architecture for $n = 4$. As suggested in the next section, several such Vector Arithmetic Modular Processors (VAMPs) might be put on a single package, depending on fabrication and processing technology, so the actual chip count may be lower. To avoid these dependencies, the discussion following will be conducted in terms of VAMPs, each capable of computing one vector result, and three together of computing a matrix-matrix multiplication.

Below, "Single Device" (single node, linear case, or tier, logarithmic) is a non-systolic configuration. "Systolic Pipeline" in the logarithmic case requires only seven nodes total on each of the forward and backward recursions, in light of the optimization discussed above for $n = 6$ applied to Figure IV.2, plus six more if the $a = b$ case is handled as in Appendix D. The algorithms would require the following number of VAMPs

VAMPs* — Propagated Variables Only ($n = 6$)				
Algorithm	Per Node	Tier, $n = 6$	Single Device	Systolic Pipeline
Linear Bkwd.	12	*	12	72
Linear Fwd.	8	*	8	48
Logarithmic Bkwd.	26	78	78	338
Logarithmic Fwd.	13	39	39	169

* *Vector Arithmetic Modular Processors — see also next section*

VAMPs — Non-Propagated Variables Only ($n = 6$)				
Algorithm	Per Node	Tier, $n = 6$	Single Device	Systolic Pipeline
Linear Bkwd.	10	*	10	60
Linear Fwd.	0	*	0	0
Logarithmic Bkwd.	10	20	20	60
Logarithmic Fwd.	0	0	0	0

The total requirements are thus

VAMPs — Totals, $n = 6$				
Algorithm	Per Node	Tier, $n = 6$	Single Device	Systolic Pipeline
Linear Bkwd.	22	*	22	132
Linear Fwd.	8	*	8	48
Logarithmic Bkwd.	36	98	98	398
Logarithmic Fwd.	13	39	39	169

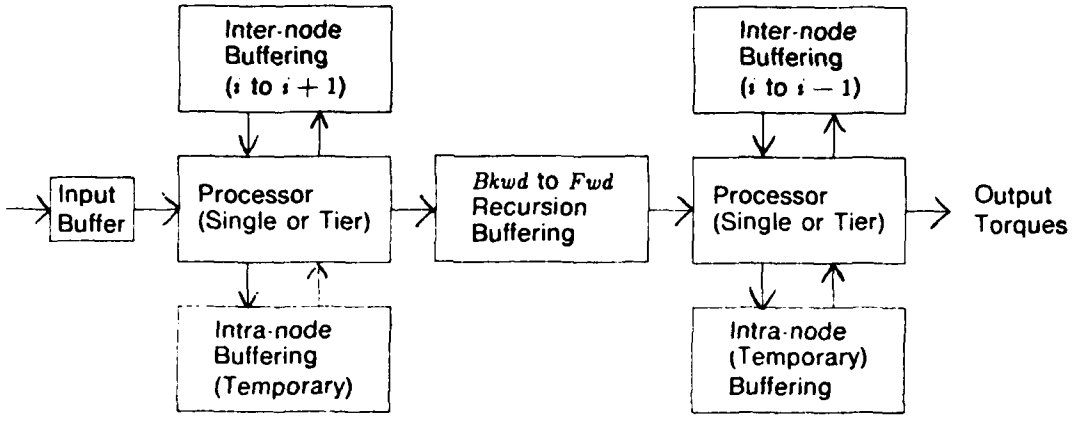


Figure VI.1a — Non-Systolic, Processors Shared, All Joints

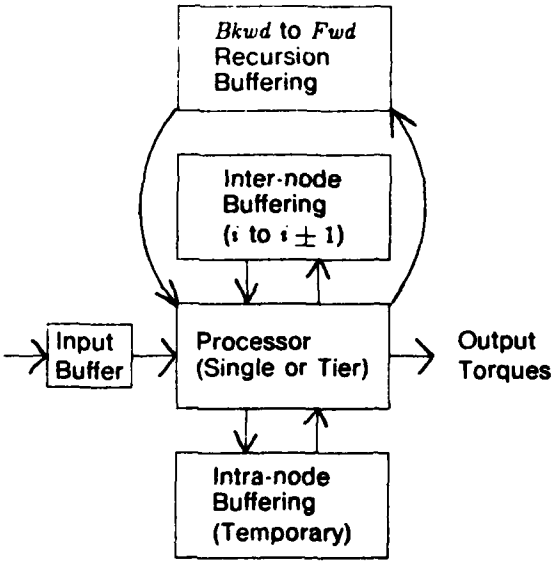


Figure VI.1b — Non-Systolic, Forward and Backward Recursion Processors Also Shared

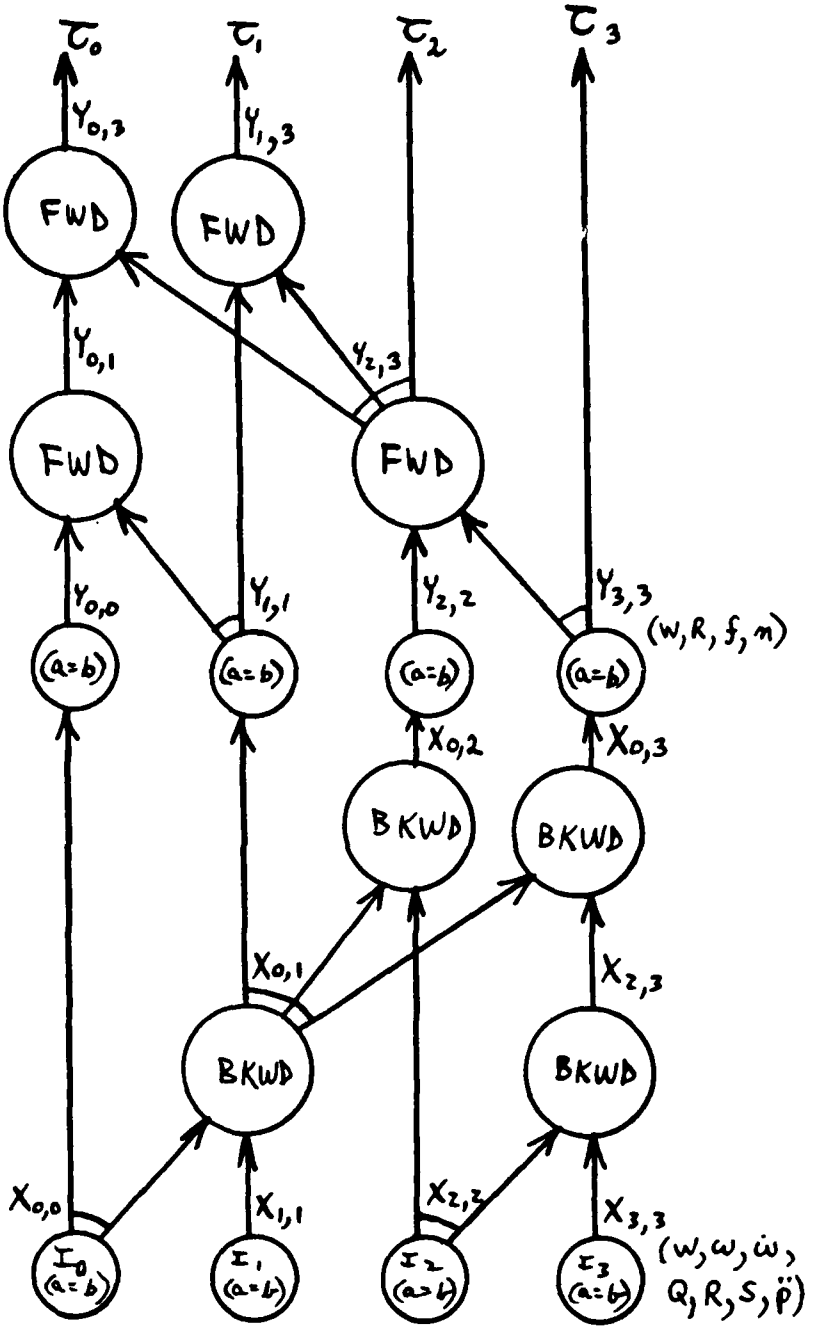


Figure VI.2 — Logarithmic Recursive Graph Structure (Both Recursions Shown)

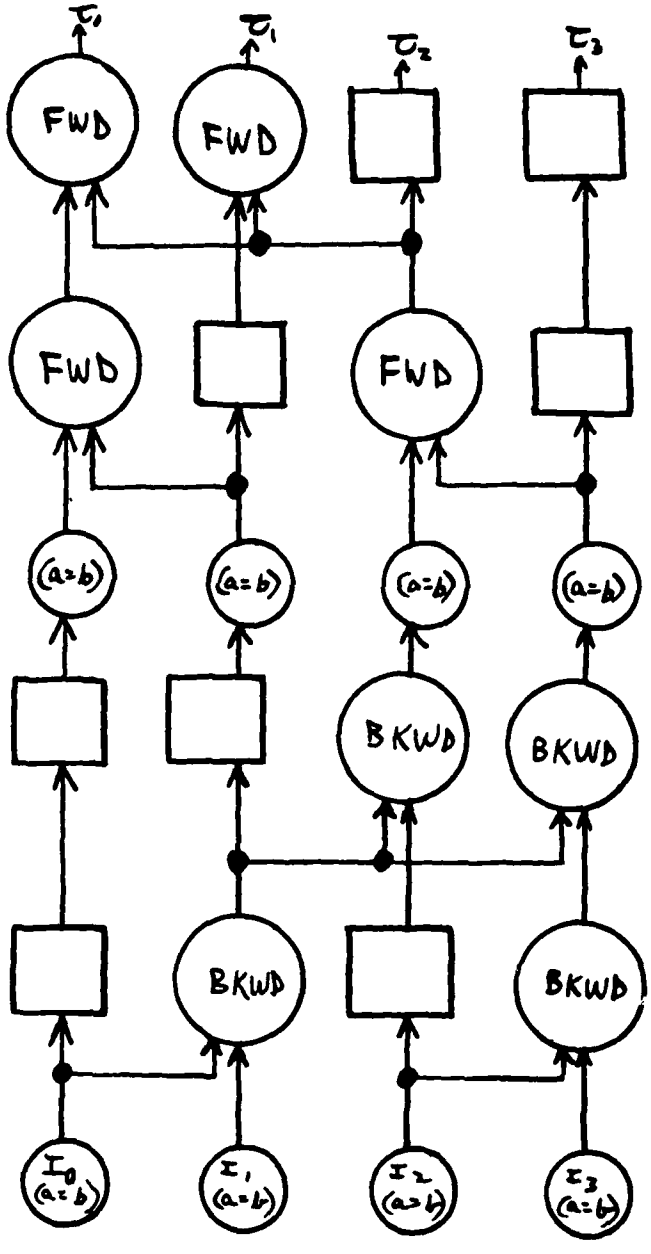


Figure VI.3 — WSI Communication Structure of Logarithmic Recursion (Rectilinear Array)

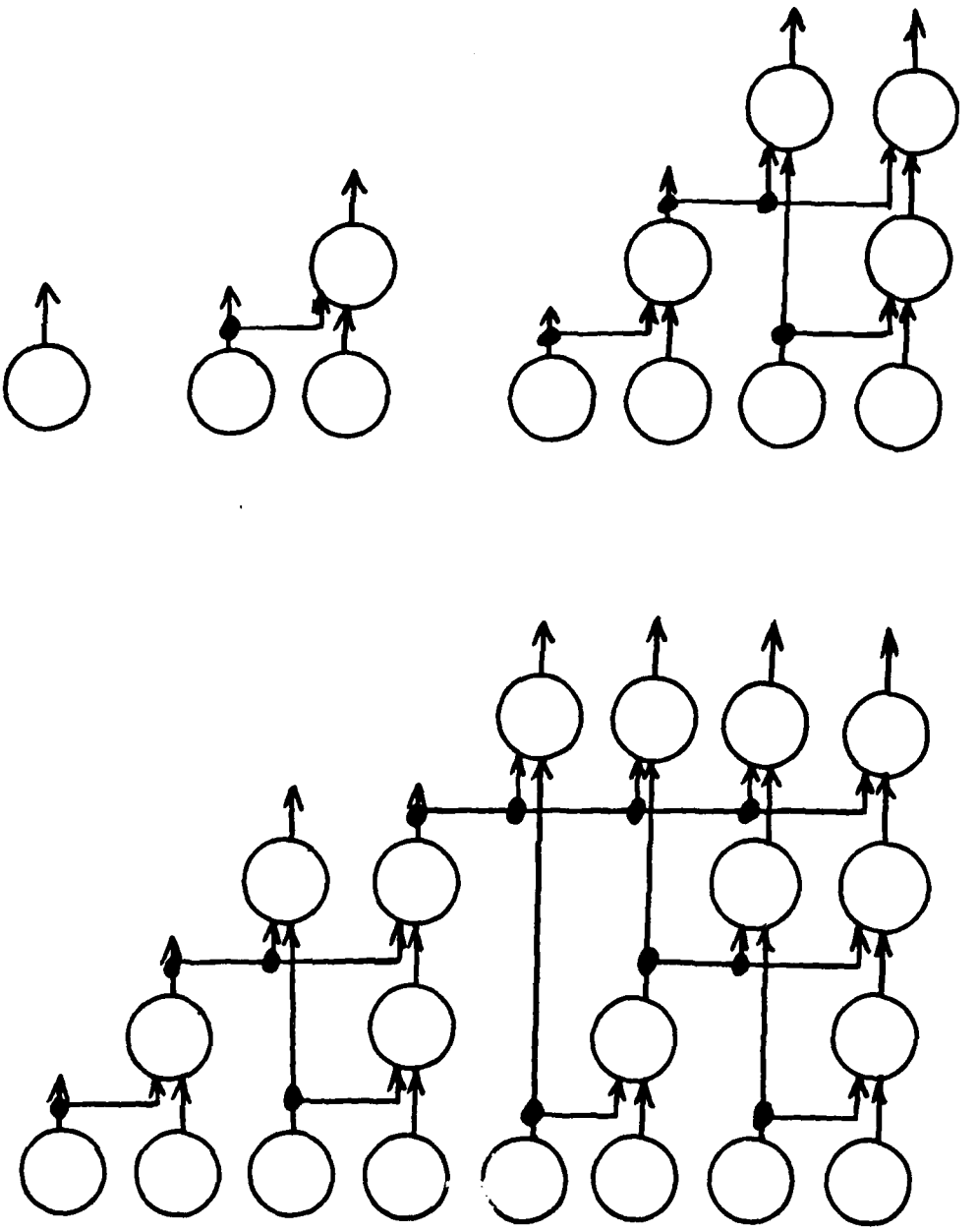


Figure VI.4 — Regularly Extensible Logarithmic Recursive Graph

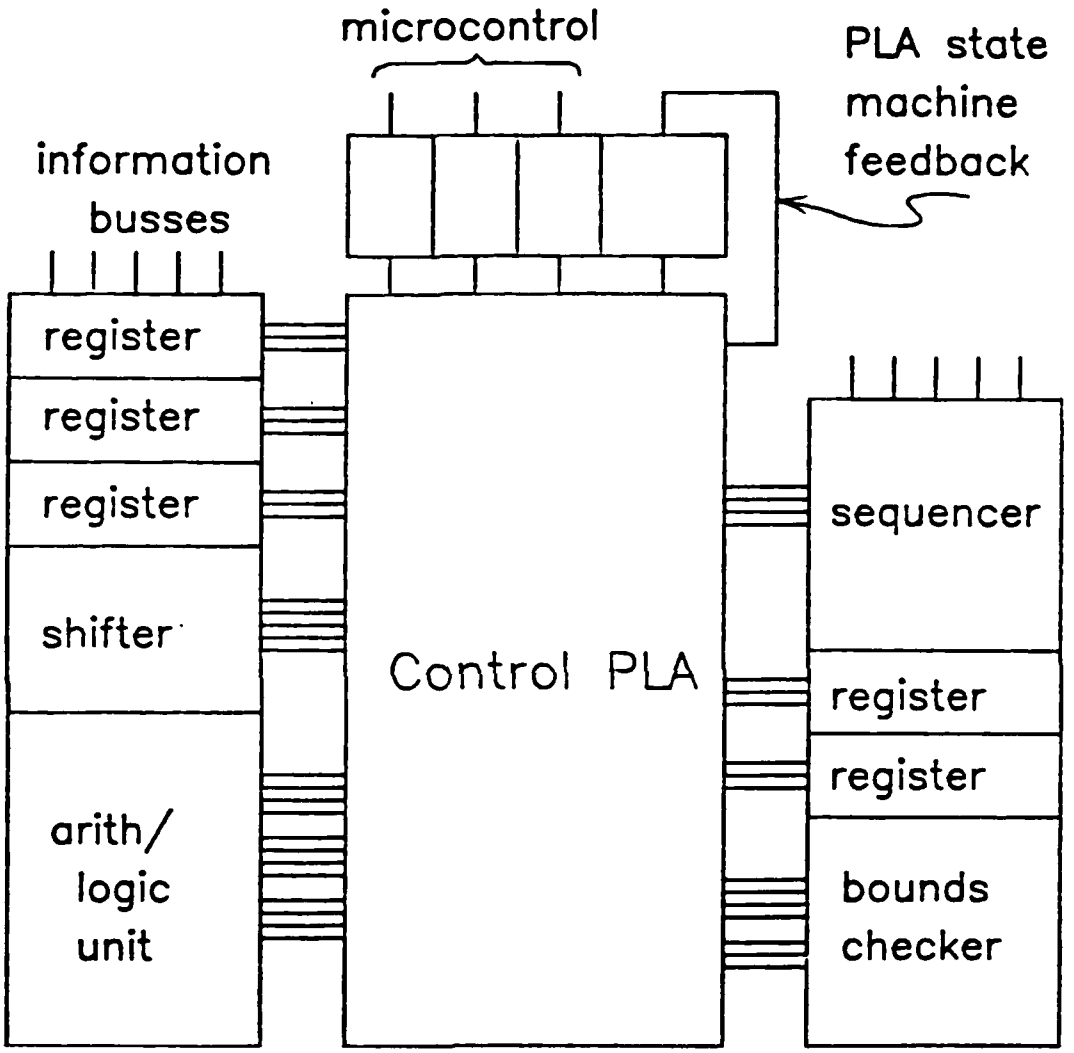


Figure VI.5 — Datapath Chip Structure

7. A ROBOT CHIP

Having seen that the global physical requirements are not excessive, it becomes interesting to look more closely at VLSI chip structures capable of physically implementing the computation. It is clear from the discussion above that we require a chip capable of performing general matrix-vector arithmetic. Within this section we investigate the computation and control architecture supporting this requirement. We will sketch one architecture suitable for our purposes, though of course there are others. As in the previous section we imagine that any actual physical implementation will differ in many particular details from those presented here, but not by so much as to render the general conclusions inapplicable.

Obviously such a chip will be more broadly applicable than just the inverse dynamics computation, however, and we will seek to maintain a high degree of flexibility in our implementation. In particular, we will seek to insure that:

(a) all chips used be identical, or at least interchangeable, so that only one chip-type is required,

(b) the particular operation which any chip performs be programmable,

(c) the length of the basic time cycle, and the points during the cycle when operands are read or the result written to busses, and which busses, be programmable,

(d) the host computer be able to dynamically reprogram (b) and (c) at will by simply writing the appropriate values to the n input buffers, in exactly the way data is written to the device, and

(e) the control flexibility of (a) through (d) must not use any communication wires or buffering other than those already required to support the computation and discussed in the previous section.

Later in the section we will consider modifications to the bussing scheme discussed in the preceding section. By relaxing (e), we will sketch an architecture whereby

(f) the sources of the operands which any particular chip accepts are programmable (as in (d)), and

(g) a limited error-correcting capability may be incorporated, so that correct computation can often be automatically continued following individual chip failures.

It should be explicitly observed, however, that the architecture presented is incapable of matrix inversion or similar operations (e.g., Gaussian elimination). It would also be useful to incorporate some of the common trigonometric functions and inverses, as well as square root. These capabilities are required for many interesting applications, but require much more complex control and condition testing than we need for basic matrix-vector arithmetic.

The basic unit of most matrix-vector arithmetic is the vector dot (inner) product. Matrix-vector multiplication is accomplished by forming the dot product of each of the rows of the matrix with the vector, and matrix-matrix multiplication by forming the dot product of each row of the first matrix with each column of the second. Other familiar operations (vector cross (outer) product, vector addition, etc.) may be performed with different sequencing of the basic dot product hardware.

Our strategy will be to compose a primitive module capable of computing one coordinate of the result (i.e., one vector dot product). If we then group several such primitive modules together, along with suitable control, we can implement all necessary matrix-vector operations. The number of primitive modules per chip is a design decision; the control mechanisms we develop support different choices. Nine primitive modules are sufficient for a matrix-matrix multiplication, and three for a matrix-vector multiplication. The basic architecture established, we display control sequences implementing the various operations. Next we embed the mechanism in a timing structure governing when in the cycle operands are read and the result computed and written. The internal structure explicated, we discuss how the control information might be loaded under programmed control from the host. Finally, we indicate how the bus implementation of chip interconnect might be extended to allow different algorithms to be dynamically programmed, and a limited error-correcting capability.

As shown in Figure VI.5, a datapath[3] is a bus-oriented architecture composed of stacked computational elements, busses running through them, and a centralized control. Typically a primitive cell, one bit wide, is replicated in the horizontal direction to the

width (in bits) of the datapath, yielding one computational element (such as a register or adder). For floating-point arithmetic this is modified slightly so that the mantissa and the exponent circuitry are replicated separately. Different computational elements are then stacked vertically to form the datapath. One or more busses run vertically through each cell, and these may be connected by abutment to vertically adjacent elements forming vertical global busses. Control lines run horizontally through each cell, control being frequently generated by a Programmable Logic Array (PLA) on the side.

For datapath elements we assume the following: registers, adders, and multipliers. (Other elements, such as comparators and dividers, would also be needed to implement matrix inversion or Gaussian elimination.) Each element "talks" to both busses and to both vertically adjacent (abutting) neighbors. Thus each element can load operands and dump results to and from the busses and adjacent neighbors. Additionally, in order to facilitate serial communication between chips and minimize wires, we assume that the registers are shift registers, and can shift operands in or results out. We observed while developing upper bounds in the preceding section that the "vector busses three scalars wide" might actually be implemented as a single multiplexed wire in order to minimize wire-count. "Operand" below will typically mean the three scalar values which make up a vector, unless context clearly indicates otherwise. (All data transfers *within* a module are done in parallel, of course.)

Thus the basic cycle will be:

(a) start cycle,

(b) load first operand from off-module, a certain delay after the cycle starts,

(c) load second operand from off-module, a second certain delay after cycle-start,

(d) compute result,

(e) dump result to off-module, another certain delay after cycle-start,

(f) delay until cycle ends, another certain delay after cycle-start, and

(g) go to (a).

As noted above, the basic cycle length of the inverse dynamics algorithm considered herein

is $MV + VA = 4$ *Flops*, though other algorithms will have other basic cycles.

Consider the floor-plan for a primitive module, shown as a block diagram in Figure VII.1. Assuming that the operands have been properly loaded, it is easy to see that it can compute one coordinate of any of the matrix-vector operations listed at the end of Tables III.1 and V.1. (Note that to enable the vector cross product (VC), the two multipliers are placed between the two register pairs NOT corresponding to the result component calculated by the primitive module.) The sequences of data transfers and computations required for each are given in Table VII.1. Only two global busses are required, and that the number of multipliers has been reduced from three to two (a consequence of the 1 *Mult* = 1 *Addn* = 1 *Flop* timing assumption, as noted in the previous section).

Several of these primitive modules will operate concurrently to produce a matrix or vector result; since our vectors are in $\mathcal{R}^3$, this number will normally be a multiple of three. As in the previous section, we will assume groups of three primitive modules. Figure VII.2 shows a floor-plan block diagram, together with control and (multiplexed) off-module communication (not shown is the ROM or RAM program storage). Such a device would be capable of one matrix-vector multiplication, or the three coordinates of any supported vector-vector operation. For purposes of discussion, we will take this Vector Arithmetic Modular Processor (VAMP) as our unit of computation.

The VAMP operand inputs are wired (perhaps through intermediate delay buffers as discussed in the preceding section) to the result output of the VAMP computing that operand. By synchronizing between source and destination VAMPs the respective dump and load delays after cycle-start (recall that we are willing to assume a global clock and reset), intermediate results can be passed to successive operators.

Serial off-module communication between VAMPs might permit several VAMPs to share the same package by substantially reducing pin-out. (The trade-off is that it will increase communication time, perhaps requiring the insertion of additional delays.) Though the physical size is heavily dependent on choice of process technology, width of the data word in bits, and other design parameters, we can make some initial estimates of total package count based solely on pin-out (with the understanding that these will likely be revised upwards depending on design parameters and the particular fabrication process). In most

process technologies, each VAMP would require external connections to **Power**, **Ground**, **Clock-1**, **Clock-2**, **Global-Reset**, **Operand-A**, **Operand-B**, and **Result**. If each VAMP is wired separately, a nominal 40-pin package would provide pin-out sufficient for five VAMPs to support these eight signals. (Due to process defects, of course, they must be individually tested and faulty VAMPs discarded.) If a single **Clock** input is converted to **Clock-1** and **Clock-2** (or more if needed, e.g. **Pre-Charge**) on-chip and distributed to each VAMP, and if **Power**, **Ground**, and **Global-Reset** are also brought in on one pin each and distributed, then a nominal 40-pin package would provide pin-out sufficient for 12 VAMPs. As was seen in the preceding section, 11 such 40-pin packages would implement the full systolic pipeline for the linear case of $n = 6$.

Alternatively the other extreme can be chosen, and each VAMP packaged individually. Each VAMP could be put in an 8-pin package, resulting in very small cheap chips. This would have the further advantage of not requiring innovative packaging.

The control must specify the delays associated with communication and computation of the result, and which operation to perform. This is shown schematically in Figure VII.3. The delays are easily implemented by counters and comparators. A Cycle-Counter is zeroed at each cycle-start and incremented at each *Flop*. The delay associated with each register load/dump is compared to the Cycle-Counter and the register loaded on equality. By insuring that cycle-start occurs synchronously on all chips, communication timing between operand sources and destinations can be coordinated programmatically. In turn, in virtue of accepting a global clock signal we can insure simultaneous global cycle-start (e.g., perhaps by bringing **Global-Reset** to **True** for one clock period only at the beginning of each cycle). This globally resets counters to zero, and insures that accidentally dropping a bit or missing a clock-tick does not throw a module permanently out of step.

It is sufficient if the device is able to execute operations from only a small fixed repertoire, viz. those at the end of Tables III.1 and V.1. This being the case, the microcode instructions needed for each item in the repertoire may be stored on-module in Read-Only-Memory (ROM). The operation to be performed can then be specified as a pointer into ROM, e.g. as the high-order address bits or as an index pointer.

If we were designing a very general machine we would certainly want the microcode

to be loaded under program control from the host. Though such capability is not needed for most matrix-vector algorithms, we observe that Random-Access-Memory (RAM) could be loaded in much the same way as we will load control information. The pointer into ROM would then be augmented with an indicator bit selecting RAM or ROM, and all would proceed as sketched for the ROM-only case.

Thus we see that communication and control can be effected by loading three registers with delays for the comparators and one register with a ROM pointer. These four values will be referred to below as the module's "program". Next, consider how these control registers might be loaded under programmed control from the host. (For a prototype version one might simply bring the control in from off-chip, each control register bit corresponding to a pin wired by jumpers to either power or ground.)

Since the same wires are to be used for both data and control, the chips must have a way to distinguish which is which. Though it would be possible to tag each word with a control bit, it is preferable to use the **Global-Reset** signal. In normal operation (Data Mode) **Global-Reset** is (for example) brought **True** for exactly one clock-tick to mark cycle-start, then immediately returned to **False**. If instead it is kept **True**, all chips enter Program Mode. While in Program Mode they will use the same wires in much the same way, but treat the words as program data rather than computational data. In Program Mode, **Global-Reset** is (for example) brought **False** for exactly one clock-tick to mark program-cycle-start, then immediately returned to **True**.

The system must be programmable from any conceivable system state, in particular from random or partially programmed states. Thus the delay registers must be assumed invalid. All transfers occur immediately following program-cycle-start. The host writes successive program data words to the input ports and these are clocked through the network. Each module will clock in program data as operands, operate on it as discussed below, and clock out program data as result. When the module clocks in the program data containing its own module program, that data will be loaded into its control register(s) as below. After all VAMPs have been programmed in this fashion, **Global-Reset** can be brought and kept low forcing Data Mode. Thereafter normal data computation can occur through the fully programmed network.

Thus we must insure that every VAMP is directly or indirectly accessible to the host, and that each VAMP can recognize its own program when seen. The first condition is satisfied by the connected directed acyclic graph nature of the network; any module not at least indirectly accessible to the host could never receive operands and hence could not participate in the computation.

One way to allow each module to recognize its own program is to assign to each one a unique number, "burned in" as in Programmable-Read-Only-Memory (PROM) before the chip is ever inserted into the network (a prototype might simply have jumpers wired to pins). The host then simply writes pairs of (VAMP-number, VAMP-program) to the network inputs, and these are clocked through the network. When a module recognizes its own number in Operand-A (i.e. the first operand), it loads the associated program. For example, the VAMP-number might be in the first scalar component of the vector, the VAMP-program contained in the remaining two scalars. Variations on this scheme could also allow RAM to be loaded, perhaps using the second scalar component as an address and the third as the microcode word to store there. The VAMP outputs Operand-A (unchanged) as its Result. The network can be programmed by writing, to each input port, one such number-program pair for each module in the network. Chips are interchangeable simply by changing the VAMP-numbers transmitted by the host.

We conclude this section by very briefly sketching extensions permitting a limited dynamically programmed communication system, and a limited error-correcting capability. Neither of these are required for the Inverse Dynamics algorithm, but would render the device more useful.

The communication requirements of parallel algorithms are often mostly local, with a few long-distance data transfers which must also be supported. This is the case in the Inverse Dynamics as well. The Mostly Local Bus (MLB) in Figure VII.4 is intended to support both high local band-width and sparse but necessary long-distance communication. It can be used by the host to gain programmatic control of the communication network implemented by the array of processors. Programmatic host control of the implemented communication structure (the data dependency graph), together with host control of the operation performed at each node, allows the same physical device to efficiently implement

many different algorithms without physical reconfiguration or rewiring.

Refer to Figure VII.4 (MLB). It depicts a multi-tiered bus, with some of the tiers composed of many short busses, some tiers composed of several medium-length busses, and some tiers composed of a very few long busses. Conceptually, imagine that each module is potentially connected to each bus directly in front of and back of it. (In any real implementation, small groups of modules would actually share long-distance drivers to keep the area penalty low for sparse long-distance traffic.) A communication network structure can be imposed by specifying when each module reads or writes which bus. By insuring that the destination module reads the same bus of the same tier at the same time the source module writes to that bus, any two modules which connect to the same MLB can communicate. Different source/destination pairs can share the same bus provided the communications occur at different times in the basic time cycle of the algorithm. Within the framework we have developed this means adjoining, to the registers governing at what time busses are read and written, additional registers governing which tier is targetted. These registers would be loaded exactly as already described.

In those algorithms susceptible to this architecture, most communciation will be local and can occur on the many short-interval busses, achieving high local bandwidth. The few long-distance lines exist to support sparse necessary long-distance communication, but if used too heavily the system performance degrades. This degradation can be made graceful, however, by inserting extra length into the basic cycle. This creates extra slots into which conflicting communications could be transferred. Note that this communciation structure is most suitable for algorithms having a straight-line systolic nature, such as characterizes the Inverse Dynamics. In cases where it is applicable, it permits relay-free communication with limited pin-out or bus connections, and would be most suited to WSI or systems with large numbers of processors. The relay-free character of the communication differs from schemes based on twisted n-cubes, which may require up to $O(\log n)$ relay delays between any two processors.

It would be possible to "tune" the MLB to the communication requirements of a particular algorithm (while preserving the capability to dynamically reconfigure so as to perform others) by adjusting the ratios of bus lengths in successive tiers. For illustrative purposes, Figure

VII.4 shows the number of processors served by each bus doubling at every second tier, with the even and odd tiers offset from each other so that each processor is roughly symmetric in communicative power each direction. The number of tiers thus grows as $\mathcal{O}(\log_2)$ of the number of processors. To tune the MLB to a particular algorithm, one would require that the number of busses of a given length was roughly proportional to the fraction of messages communicated a distance of approximately that length. This rough guide must be refined further if the average message length varies with distance.

Finally, we note that most of the machinery necessary to support a limited error correcting capability has been developed. This will permit several cases of gross individual chip failures to be caught and flagged, with the device automatically resuming correct computation following error correction. There are hardware error classes for which this capability will not apply, of course, some examples being a direct short between power and ground (it is difficult to automatically recover from this in any case), occassional intermittent faults, or failures in the error-correcting circuitry itself. Since the error-correcting circuitry is a small fraction of the total chip circuitry, however, the probability that it will fail first is likely also to be small.

Order the VAMPs, so that each has a unique successor and a unique predecessor except the two end VAMPs. Basically we suggest a mechanism in which each VAMP checks its successor after having been checked by its predecessor. It thereafter plays the role of its successor in the computation while its successor checks the next VAMP, and so forth. After each VAMP has been checked they return to a normal configuration, and repeat the process. If a faulty computation should be detected in the checking process, the faulty VAMP will be inhibited by its predecessor and excised from the computational structure of the device. Its predecessor will continue to play its role in the computation until the faulty component can be replaced.

In addition to those discussed above, each VAMP would also have a second set of inputs **Operand-A'**, **Operand-B'**, inputs for **Test-in**, **Test-back-in** and **Inhibit-in**, outputs for **Test-out**, **Test-back-out** and **Inhibit-out**, and input/output **Result'**. Each VAMP's **Operand-A'** and **Operand-B'** inputs are connected to the **Operand-A** and **Operand-B** inputs, and its **Result'** input/output connected to the **Result** output, of its *successor*. Its **Test-**

in and **Inhibit-in** inputs are connected to the **Test-out** and **Inhibit-out** of its *predecessor*. Its **Test-back-in** input is connected to the **Test-back-out** output of its *successor*. See Figure VII.5.

These new inputs and outputs are used as follows. During normal operation of a VAMP, **Test-in**, **Test-back-in**, **Inhibit-in**, **Test-out**, **Test-back-out** and **Inhibit-out** will all be **False**. The VAMP will load as its operands the inputs **Operand-A** and **Operand-B**, and dump its result on the output **Result**. The input/output **Result'** will be tri-stated. Eventually, the VAMP's predecessor will bring **Test-in** to **True** at the start of a cycle. The predecessor is now filling the VAMP's computational role, and the VAMP is free to check its successor. It does this by loading its operands for that cycle from the inputs **Operand-A'** and **Operand-B'**, performing the same calculation as its successor, and comparing its result to its successor's output, which is loaded through the VAMP's input/output **Result'**. (To do this, of course, it must have a second set of control registers, correctly loaded to match those of its successor.) The output **Result** is tri-stated.

If the two results do not match, then the successor is assumed to be faulty (recall that the VAMP itself has just been checked in the immediately preceding cycle). The VAMP excises its successor from the computation by bringing **Inhibit-out** to **True**. This causes its successor to tri-state its data outputs and bring its control outputs **False**, which is implemented by very simple gate logic at each pad driven directly from **Inhibit-in**. The VAMP will continue to fill its successor's role in the computation until the device is brought down for maintenance. At that time it can broadcast its VAMP-number through the net in Maintenance Mode as a fault-finder, somewhat akin to the way programs were broadcast in Program Mode, except that instead of outputting Operand A unchanged each VAMP outputs any Operand which contains a fault notification. The VAMP-numbers corresponding to fault-finders will therefore emerge from the net where the motor torques emerge in Data Mode, and the faulty VAMPs can be replaced.

If the two results match, then the successor is assumed to be not faulty. The VAMP will bring **Test-out** to **True** at the start of the next cycle, causing its successor to repeat the operation just described. Since the VAMP must now fill its successor's role, it will continue to take as its operands the inputs **Operand-A'** and **Operand-B'**, but in addition will now

dump its result on the input/output **Result'**. The output **Result** remains tri-stated.

The VAMP will return to normal operation after its input **Test-back-in** (connected to its successor's output **Test-back-out**) is brought **True** for one cycle. The input/output **Result'** is tri-stated, and its output **Test-back-out** is brought **True** for one cycle (causing its predecessor to resume normal operation in the same sequence). On the next cycle the VAMP will again load as operands the inputs **Operand-A** and **Operand-B**, and dump its result on the output **Result**.

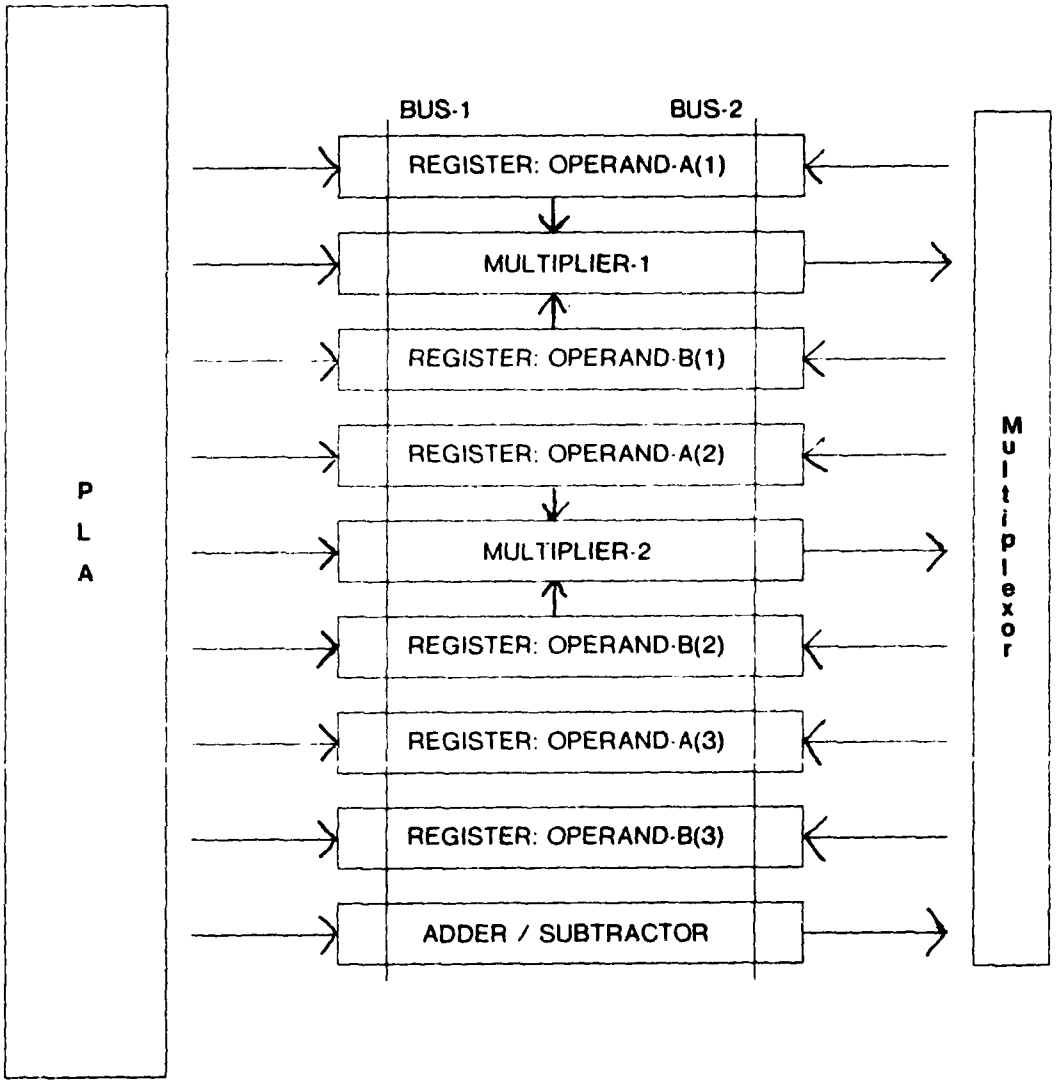
So far we have described the action of VAMPs in the middle of the order. It remains to describe the ends. The last VAMP can simply have its output **Test-out** fed back into its input **Test-back-in**. This causes the sequence to reverse when the last VAMP is reached. To check the first VAMP we must introduce an extraneous VAMP (and properly speaking, a second extraneous VAMP to check *that* one). The first extraneous VAMP can simply have its output **Test-back-out** fed back into its input **Test-in**.

Table VII.1 — Primitive Module Operation Sequencing

(refer to Figure VII.1)

(This primitive module calculates the 3rd scalar component of the result vector)

Operation	Flops	Dest.	Source
VA:	1	BUS-1	⇐ REG: OP-A-3
		BUS-2	⇐ REG: OP-B-3
		ADDER	⇐ BUS-1, BUS-2
(result in adder/subtractor)			
SV:	1	BUS-1	⇐ REG: OP-A-1
		BUS-2	⇐ REG: OP-B-3
		MULT-1	⇐ BUS-1, BUS-2
(scalar multiplier in 1st scalar component of Operand A, by convention)			
(result in multiplier-1)			
VD, MV, MM:	1	BUS-1	⇐ REG: OP-A-3
		BUS-2	⇐ REG: OP-B-3
		MULT-1	⇐ BUS-1, BUS-2
		MULT-2	⇐ REGS: OP-A-2, OP-B-2
	2	BUS-1	⇐ MULT-1
		BUS-2	⇐ MULT-2
		ADDER	⇐ BUS-1, BUS-2
		MULT-1	⇐ REGS: OP-A-1, OP-B-1
	3	BUS-1	⇐ MULT-1
		BUS-2	⇐ ADDER
		ADDER	⇐ BUS-1, BUS-2
(alternatively, adder can accumulate instead of being totally bus-driven)			
(result in multiplier-1)			
VC:	1	BUS-1	⇐ REG: OP-B-2
		BUS-2	⇐ REG: OP-B-1
		MULT-1	⇐ BUS-1, REG: OP-A-1
		MULT-2	⇐ BUS-2, REG: OP-A-2
	2	BUS-1	⇐ MULT-1
		BUS-2	⇐ MULT-2
		SUBTRACTOR	⇐ BUS-1, BUS-2
(subtractor computes BUS-1 — BUS-2)			
(result in adder/subtractor)			



(To enable VC. the two multipliers are placed between the two register pairs NOT corresponding to the result component calculated by the primitive module. Thus this example calculates the 3rd scalar component of the result.)

Figure VII.1 — Primitive Module Block Diagram

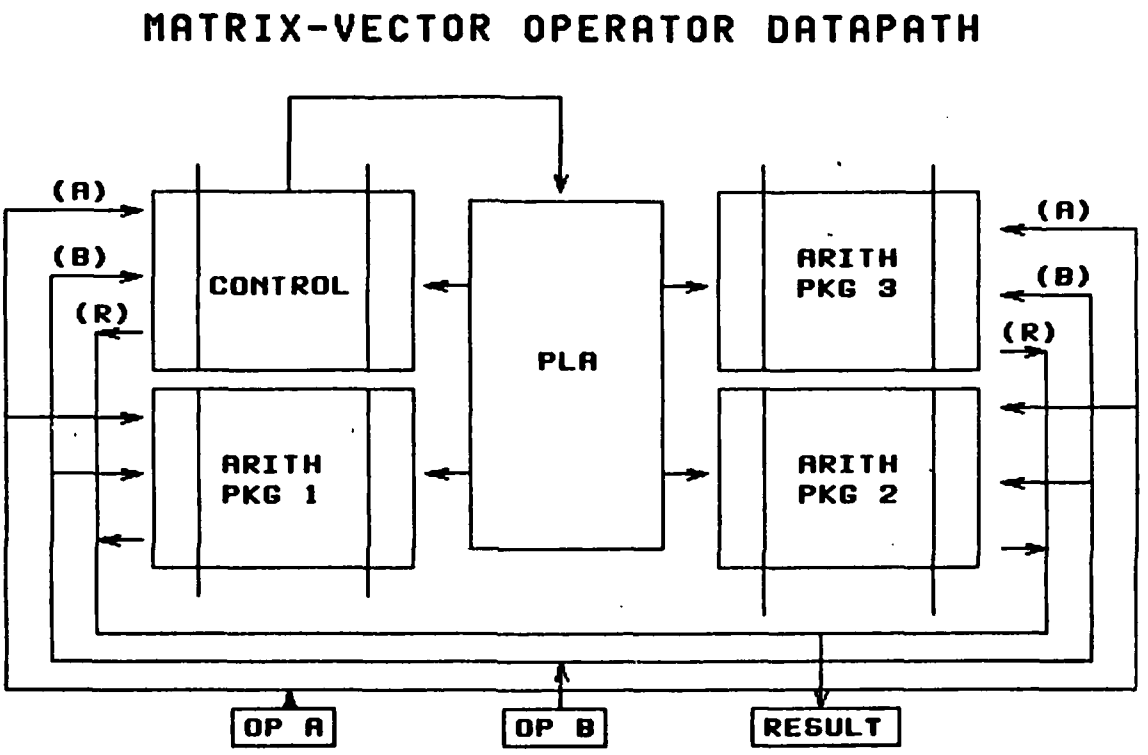


Figure VII.2 — Vector Arithmetic Modular Processor (VAMP)

CONTROL STRUCTURE

IN OPERATION, CONTROL MUST GOVERN:

1) When operand A is read in, & to which regs;
2) When operand B is read in, & to which regs;
3) When the Result is output, & from which regs;
4) What operation is performed.

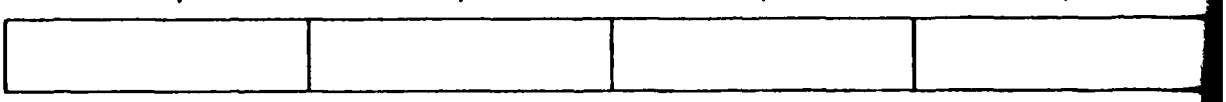
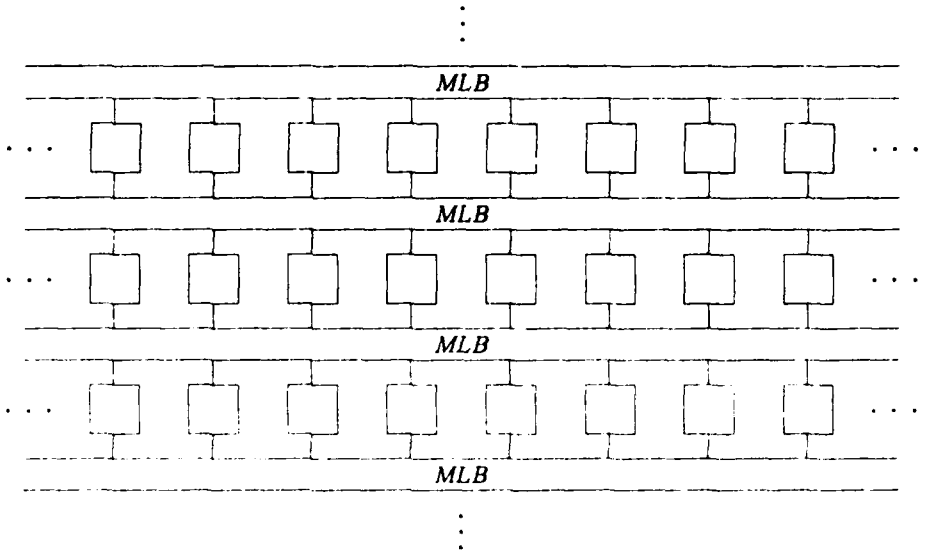
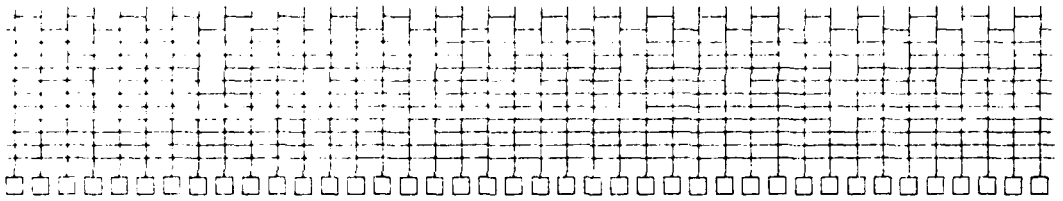


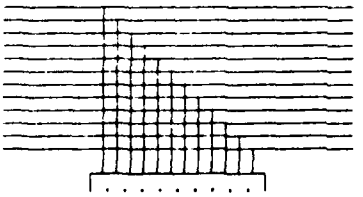
Figure VII.3 — VAMP Control Registers



Global Architecture



Detail Expansion



Fine Detail Expansion

Figure VII.4 — Mostly-Local Bus (MLB)

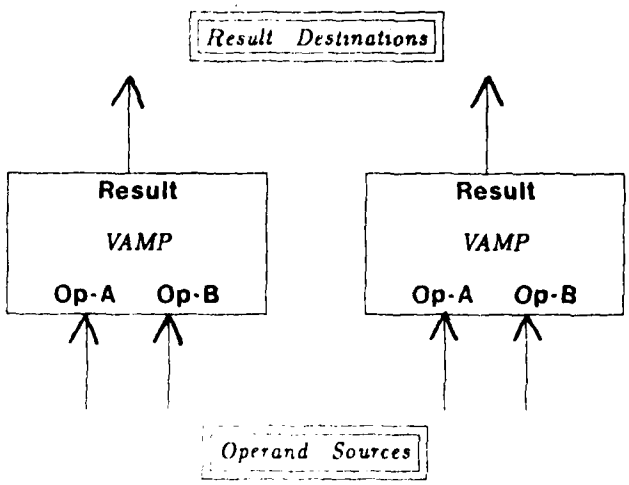


Figure VII.5a — Non-Error-Correcting Configuration

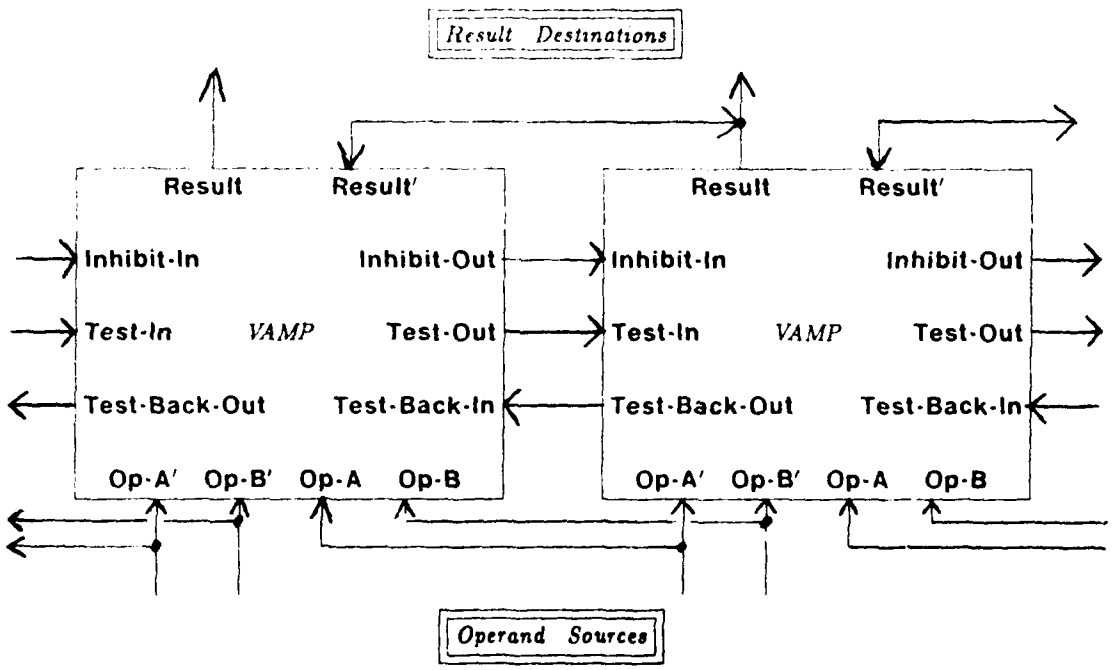


Figure VII.5b — Limited Automatic Error-Correction

8. SUGGESTIONS FOR FUTURE EXTENSIONS

We very briefly allude to, without discussing in depth, a few possible extensions to this work. If the computation can be made to run at a rate which is only I/O bound, it becomes feasible to consider "active memory" (or an I/O device) which provides the calculated torques as soon as the desired motion has been loaded in appropriate locations. Thus it may be possible to build an on-line "optimizing trajectory compiler" in which the desired motion (trajectory) for the next several time periods is pre-planned, the motor torques automatically generated, and the time sequence of necessary motor torques inspected slightly *before* the manipulator has actually arrived at the trajectory points. If the motor torques are excessive (motor or joint damage) or below the rated maximum (faster motion possible), the proposed trajectory could be modified accordingly, on-line. Also, Torre and Poggio have a result indicating that neural structures could perform an arithmetic multiplication in about a millisecond. While certainly not arguing that in fact it is done that way in the brain, observe that in principle it would be possible to compute the Inverse Dynamics in approximately real time using a suitable neural structure. We close with a few remarks concerning the possibility of generalizing the $\mathcal{O}(\log(n))$ embedding to other recursive structures.

The question of how to incorporate dynamical considerations into on-line trajectory planning is an area of open research [7]. Hollerbach[15] shows how to uniformly scale the velocity of a trajectory so as to remain within torque limits. This applies only to uniform velocity scaling, however, and we might like to be able to change the path of the trajectory or scale velocity in a non-uniform way. Bobrow, Dubowsky and Gibson[6] solve the general case of time-optimal control along a specified trajectory, but do not allow the path to be varied in space and require moderately intensive computation by the host.

Consider Figure VIII.1, which might be taken to represent a shift register of depth m. Data records pushed in at the top progresses through the shift register with timestep λ to emerge out the bottom time $m\lambda$ later.

Imagine that the shift register was interfaced to a memory board in such a way as to "look like" memory to the host, so that any of the locations could be read or written as memory. At each timestep λ the 0^{th} row would be shifted out the bottom, the whole array

shifted down one, and the host would write a new set of values at the top. At any time, of course, the host could read or write any of the array locations.

Now imagine that some of the array locations in the j^{th} row correspond to "input values" ($\{q(j\lambda), \dot{q}(j\lambda), \ddot{q}(j\lambda)\}$) and some locations correspond to "motor torques" ($\{\tau(j\lambda)\}$); and that the "torque" locations are really hard-wired (through such a dynamics box as we have described) to the "input value" locations. The box continuously computes, for each set of input values in the shift register, the corresponding torques — if the host changes an input value anywhere in the array, the torques corresponding to the new set of values automagically appear. Set λ equal to the refresh rate at which new motor torques are supplied to the manipulator. Now as each value is shifted out the bottom, imagine that a demon catches it and passes the torques to the manipulator motors — simultaneously another demon fills in a new set of input values at the top and the torques appear. We might as well have the inertial Cartesian coordinates and velocities of each link endpoint appear too; since they are manifestly simpler to calculate from the same input data as the inverse dynamics, the fractional cost to include them is small.

This now becomes a fairly explicit representation of many interesting characteristics of the manipulator trajectory for the next m time periods. If the host inspects the values but makes no changes, those m values will be shifted to the manipulator, one by one; and that will define the path the manipulator will follow for the next m periods. Alternatively the host may change one or more values causing the corresponding changes in the torques; the manipulator would then follow the revised course. This arrangement might be useful in helping to solve the problem of how to incorporate dynamical considerations into on-line trajectory planning, acting to help optimize a crude trajectory generated by a higher-level planner.

One extension to this basic idea would be to use our box to also calculate the input value joint velocities and positions ($\{q(t), \dot{q}(t)\}$) directly from the acceleration profile ($\{\ddot{q}(t)\}$), rather than having them set directly by the host. This would avoid the embarassing possibility of (e.g.) a trajectory requiring an instantaneous step discontinuity in manipulator position, as well as reduce computational demands on the host. Another extension would be to have several such shift registers; one could thereby sweep out an envelope around the proposed

trajectory, exploring in parallel possible futures and interpolating between them.

Another interesting area is the overlap with human psychology and neurophysiology. Torre and Poggio[41],[22] have shown the theoretical possibility that a neuron could perform an analog multiplication in its dendritic branches within about a millisecond (this capability was originally postulated to be necessary in order to explain certain aspects of visual processing). The analysis treats the dendritic branches according to membrane theory in passive RC cables. Where g_1 and g_2 are inputs they are able to produce a term proportional to $(g_1 - \alpha g_1 g_2)$. By additionally connecting the g_1 input appropriately to a side branch on the dendritic pathway to the axon it is possible to show theoretical cancellation of the linear term. Analog additions may be performed as in classical circuit theory (given appropriate arrangements of the dendrites).

Thus, given the time bounds on the formulations developed above, the nervous system might be capable of performing the inverse dynamics calculation in something approximating real time. Since the computations are performed in analog by biological components one necessarily expects them to be "dirty", i.e. contaminated with noise, inaccuracies, and other errors. If one postulates a large number of such inaccurate devices performing the same calculation and averaging the result, however, from fundamental statistical properties one may show that the resulting calculation can be made arbitrarily accurate by taking the number of devices arbitrarily large. Also, though the formalisms were developed for a single chain of length n, the fact that the time complexity increases only as $\mathcal{O}(\log(n))$ suggests the possibility that one might control other large systems involving many degrees of freedom without paying an exorbitant penalty in real elapsed time. Hollerbach and Flash[17] have performed experiments investigating the possibility that human subjects perform some sort of dynamic scaling in planning planar arm trajectories. Many associated questions immediately arise, of course, such as how the nervous system learns to perform the computation; or, if it is hard-wired, how the system learns the parameters; and so forth. We do not wish to engage in this debate, but only to point out that the computational aspects are tractable.

Finally, it seems likely that the process of embedding a serial linear-time recursive algorithm in a parallel logarithmic-time algorithm is generalizable, certainly at least within the context of associative ring operators. We saw that several basic properties were exploited

in our analysis above, including the associativity of the ring operators and the capability to order the recursive variables in time according to data dependencies.

The basic strategy followed in Section IV was to expand the closed-form non-recursive formula for $X_{a,b}$ into an equivalent expression involving two similar formulae for $X_{a,k}$ and $X_{k+1,b}$. This was taken to be the combining form for $X_{a,b}$. In making the expansion, it was also necessary to expand and re-group expressions for the dependent variables in terms of their combining forms. We speculate that the associativity of the operators permitted the necessary expansion of the dependent variables. Linearity is not necessary, as it is possible to devise a combining form for the (non-linear) $\left(\frac{1}{A+B}\right)$ from expressions for $\left(\frac{1}{A}\right)$ and $\left(\frac{1}{B}\right)$:

$$\left(\frac{1}{A+B}\right) = \frac{\left(\frac{1}{A}\right)\left(\frac{1}{B}\right)}{\left(\frac{1}{A}\right)+\left(\frac{1}{B}\right)}$$
$$= \left(\frac{1}{A}\right) \otimes \left(\frac{1}{B}\right).$$

Thus it might be possible to devise logarithmic-time parallel algorithms from linear-time serial ones even if scalar division is involved, and perhaps other non-linear (non-ring) operators. Of particular interest would be a general mechanism for logarithmic embedding of linear algorithms involving matrix inversion (or solutions of simultaneous linear equations, e.g. Gaussian elimination) at each step. This would render a much wider class of algorithms accessible to logarithmic-time techniques, e.g. perhaps the inverse kinematics or the direct (integral) dynamics.

In Appendix A it is noted that the ability to order the recursive variables according to data dependencies

$$X_i \succeq Y_i \succeq Z_i \succeq \ldots$$

where we define

$$X_i \succeq Y_i \text{ iff } Y_i \text{ does not depend on } X_i \text{ for any } j$$
$$\text{iff } \partial Y_i / \partial X_j \equiv 0$$

guaranteed minimal satisfiability of the relative offset inequalities. We speculate that this condition might also be necessary to form logarithmic-time combining forms as well. This may arise since in devising the combining forms it was necessary to apply previously-deduced combining forms to break apart the constituent dependent variables. Lacking this condition,

one may at least imagine a case in which deriving the combining form for $X_{a,l}$ requires expanding the combining form for $Y_{a,b}$, while deriving the combining form for $Y_{a,l}$ requires doing the same for $X_{a,b}$. The ability to order the recursive variables by data dependencies insures that at least this particular deadlock cannot arise. A more formal demonstration of the applicability of this condition would be useful.

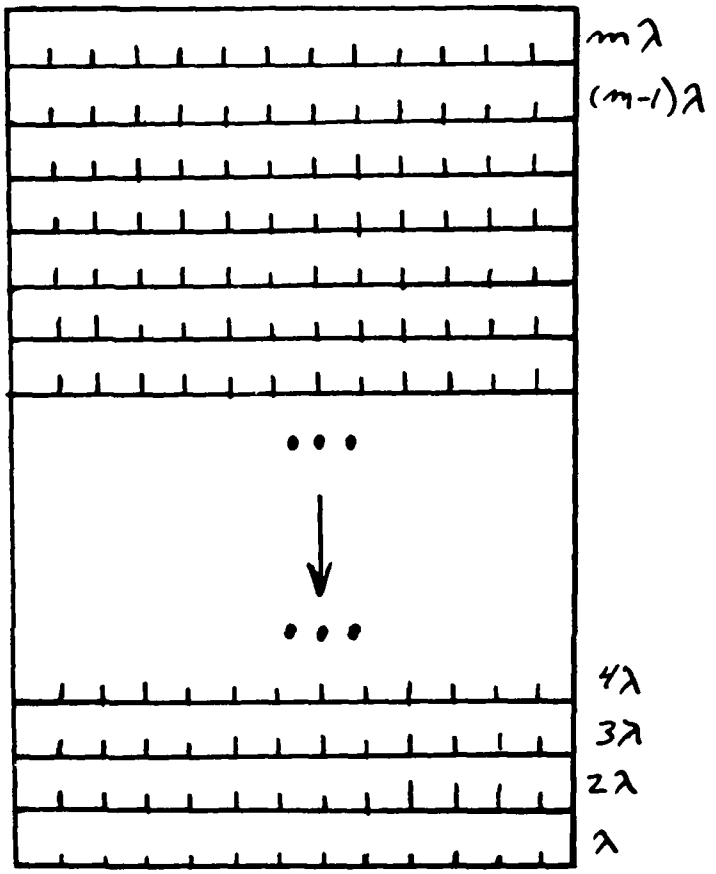


Figure VIII.1 — Fall-Through Memory Shift Register

9. CONCLUSIONS

We have shown that considerable time savings accrues from embedding the inverse dynamics calculation in a parallel computation. A parallel-time algorithm with time complexity only logarithmic in the number of joints has been derived. Hardware necessary to implement such parallel algorithms has been considered, and the requirements shown to be substantial but not excessive. Using the concepts developed, it should be possible to devise a device capable of implementing the calculation at a speed primarily bounded by the input/output requirements which the algorithm imposes on the host. We have sketched speculative extensions to this work in the areas of on-line trajectory planning, psychology and neurophysiology, and parallel algorithm theory.

REFERENCES:

[1] Albus, J. S.; "A New Approach to Manipulator Control; the Cerebellar Model Articulation Controller (CMAC)", *Journal of Dynamic Systems, Measurement, and Control, Vol. 97*, pp. 270-277, 1975.

[2] Albus, J. S.; "Data Storage in the Cerebellar Model Articulation Controller (CMAC)", *Journal of Dynamic Systems, Measurement, and Control, Vol. 97*, pp. 228-233, 1975.

[3] Barrett, W. A.; Rogers, B.; Lathrop, R.H.; Kuchinsky, A.; "An Extensible Datapath Generator", Intl. Conf. on Computer-Aided Design (IEEE-ICCAD'83), Santa Clara, Ca., Sept 12-15, 1983, pp. 1-2.

[5] Besant, C. B.; "A Multiple Microcomputer Robot Controller", *Trans. Amer. Nuclear Soc.*, Vol. 43, pp. 751-752.

[4] Bejczy, A. K.; *Robot Arm Dynamics and Control*, Technical Memorandum 33-669, Jet Propulsion Laboratory, February 1974.

[6] Bobrow, J.E.; Dubowsky, S.; Gibson, J.S.; "On the Optimal Control of Robotic Manipulators with Actuator Constraints", *Proc. American Control Conf.*, June 22-24, 1983, San Francisco, Ca., pp. 782-787.

[7] Brady, J.M.; Hollerbach, J.M.; Johnson, T.L.; Lozano-Perez, T.; Mason, M.T.; *Robot Motion: Planning and Control*, M.I.T. Press, Cambridge, Mass., 1983.

[8] Carlisle, B.H.; "Micros and Minis in CNC", *Machine Design*, Vol. 54, No. 19, August 26, 1982, pp. 66-71.

[9] Cook, G.E.; Levick, P.C.; Welch, D.; Wells, A.M., Jr.; "Distributed Microcomputer Control of an Automated Arc Welding System", *Conf. Record of Industry Applications Society Annual Meeting (IEEE-IAS-1982)*, October 4-7, 1982, San Francisco, Ca., pp. 1296-1302.

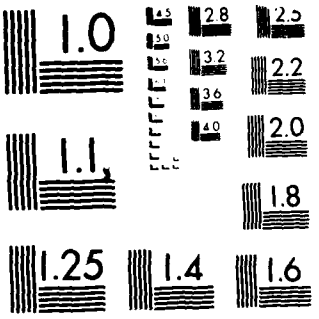
[10] Denavit, J.; Hartenberg, R.S.; "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices", *ASME Journal of Applied Mechanics*, pp.215-221, June 1955.

[11] Dubowsky, S.; "On the Adaptive Control of Robotic Manipulators: The Discrete-Time Case", *1981 Joint Automatic Control Conference (JACC)*, June 17-19, 1981, Charlottesville, Va., pp. TA-2B/1-9, vol. 1.

[12] Goshorn, L.A.; "A Single-Board Approach to Robotic Intelligence", *Computer Design*, Vol. 21, No. 11, November 1982, pp. 1 3-201.

AD-A142 515 PARALLELISM IN MANIPULATOR DYNAMICS REVISION(U) 2/2
MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB R H LATHROP DEC 83 AI-TR-754-REV
UNCLASSIFIED N00014-80-C-0505 F/G 6/4 NL

END
DATE FILMED
8-84
DTIC



[13] Guo, T.H.; Koivo, A.J.: "Microprocessor Implementation of an Adaptive Controller for Robotic Manipulators", *Proc. IEEE Computer Soc. Conf. on Pattern Recognition and Image Processing*, June 14-17, 1982, Las Vegas, Nevada, pp. 641-646.

[14] Gupta, P.; "Multiprocessing Improves Robotic Accuracy and Control", *Computer Design*, Vol. 21, No. 11, November 1982, pp. 169-176.

[15] Hollerbach, J.M.; "Dynamic Scaling of Manipulator Trajectories", accepted to *Journal of Dynamic Systems, Measurement, and Control*; also available as M.I.T. Artificial Intelligence Laboratory Memo No. 700, Massachusetts Institute of Technology, Cambridge, Mass., January, 1983.

[16] Hollerbach, J.M.; "A Recursive Formulation of Manipulator Dynamics", *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-10, No. 11, pp. 730-736, 1980.

[17] Hollerbach, J.M.; Flash, T.; "Dynamic Interactions Between Limb Segments During Planar Arm Movement", *Biol. Cybernetics*, Vol. 44, pp. 67-77, 1982.

[18] Hollerbach, J.M.; Sahar, G.; "Wrist-Partitioned Inverse Kinematic Accelerations and Manipulator Dynamics", *Intl. Journal of Robotics Research*, Vol. 2, No. 4, Winter 1983, forthcoming; also available as M.I.T. Artificial Intelligence Laboratory Memo No. 717, Massachusetts Institute of Technology, Cambridge, Mass., April, 1983.

[19] Kahn, M.E.; *The Near-Minimum-Time Control of Open-Loop Articulated Kinematic Chains*. Stanford Artificial Intelligence Laboratory Memo No. 106, Stanford University, Stanford, Ca., December 1969.

[20] Kane, R.K.; Levinson, D.A.; "The Use of Kane's Dynamical Equations in Robotics", *Intl. Journal of Robotics Research*, Vol. 2, No. 3, Fall 1983, pp. 3-21.

[21] Klein, C. A.; Wahawisan, W.; "Use of a Multiprocessor for Control of a Robotic System", *Intl. Journal of Robotics Research*, Vol. 1, No. 2, Summer 1982, pp. 45-59.

[22] Koch, C.; Poggio, T.; Torres, V.; "Retinal Ganglion Cells: A Functional Interpretation of Dendritic Morphology", *Phil. Trans. R. Soc. Lond.* Vol. 298 B 1090, pp. 227-264, July 1982.

[23] Kopacek, P.; "Microcomputer Control of Manipulators and Assembling Machines", *Control Science and Technology for the Progress of Society, Proc. of the 8th Triennial World Congress of the Intl. Federation of Automatic Control*, Kyoto, Japan, August 24-28, 1981, pp. 1897-1901.

[24] Kuo, M. H.; "Distributed Computing on an Experimental Robot Control System", *IEEE 1981 IECI Proc., Applications of Mini and Microcomputers*, November 9-12, 1981, San Francisco, Ca., pp. 330-335.

[25] Lathrop, R. H.; *Parallelism in Arms and Legs*, S.M. Thesis, Massachusetts Institute of Technology, December 1982.

[26] Lee, C. S. G.; Mudge, T. N.; Turney, J. L.; "Hierarchical Control Structure Using Special Purpose Processors for the Control of Robot Arms", *Proc. IEEE Computer Soc. Conf. on Pattern Recognition and Image Processing*, June 14-17, 1982, Las Vegas, Nevada, pp. 634-640.

[27] Luh, J. Y. S.; "Scheduling of Distributed Computer Control Systems for Industrial Robots", *Distributed Computer Control Systems 1981, Proc. Third IFAC Workshop*, August 15-17, 1981, Beijing, China, pp. 85-102.

[28] Luh, J. Y. S.; Lin, C. S.; "Scheduling of Parallel Computation for a Computer-Controlled Mechanical Manipulator", *IEEE Trans. on Systems, Man, and Cybernetics*, Vol. SMC-12, No. 2, March/April 1982, pp. 214-234.

[29] Luh, J. Y. S.; Walker, M. W.; Paul, R. P. C.; "On-Line Computational Scheme for Mechanical Manipulators", *Journal of Dynamic Systems, Measurement, and Control, Vol 102*, June 1980, pp. 69-76; Errata Correction September 1980, pp. 173.

[30] Luh, J. Y. S.; Walker, M. W.; Paul, R. P. C.; "Resolved-Acceleration Control of Mechanical Manipulators", *IEEE Transactions on Automatic Control*, Vol. 25, No. 3, June 1980, pp. 468-474.

[31] Martin, H. L.; Satterlee, P. E.; Bolfing, B. J.; "Distributed Digital Processing for Servo-Manipulator Control", *Trans. Amer. Nuclear Soc.*, Vol. 43, pp. 752-753.

[32] Mudge, T. N.; "Special Purpose VLSI Processors for Industrial Robots", *Proc. COMPSAC-81, IEEE Computer Soc. 5th Intl. Computer Software and Applications Conference*, November 18-20, Chicago, Ill., pp. 270-271.

[33] Mudge, T.N.; Turney, J.L.; "Unifying Robot Arm Control", *Conf. Record of Industry Applications Soc. Annual Meeting, (IEEE-IAS-1982)*, October 4-7, 1982, San Francisco, Ca., pp. 1315-1324.

[34] Orin, D. E.; McGhee, R.B.; Vukobratovic, M.; Hartoch, G.; "Kinematic and Kinetic Analysis of Open-Chain Linkages Utilizing Newton-Euler Methods", *Mathematical Biosciences*, Vol. 43, No. 1/2, pp. 107 -130, February 1979.

[35] Paul, R.C.; *Modeling, Trajectory Calculation and Servoing of a Computer Controlled Arm*, Stanford Artificial Intelligence Laboratory Memo No. 177, Stanford University, Stanford, Ca., September 1972.

[36] Rafauli, R.; Sinha, N.; Tlusty, J.; "A Distributed Microprocessor Control System for an Industrial Robot", *IEEE 1981 IECI Proc., Applications of Mini and Microcomputers*,

November 9-12, 1981, San Francisco, Ca., pp. 319-323.

[37] Raibert, M.H., "A Model for Sensori-Motor Control and Learning", *Biological Cybernetics*, Vol. 29, pp. 29-36, 1978.

[38] Raibert, M.H.; Horn, B. K. P.; "Manipulator Control Using the Configuration Space Method", *The Industrial Robot*, Vol. 5, No. 2, June 1978, pp. 69-73.

[39] Shin, K.G.; Malin, S.B.; "Dynamic Adaptation of Robot Kinematic Control to its Actual Behavior", *Proc. Intl. Conf. on Cybernetics and Society (IEEE)*, October 26-29, 1981, Atlanta, Ga., pp. 420-427.

[40] Silver, W.; "On the Equivalence of Lagrangian and Newton-Euler Dynamics for Manipulators", *Robotics Research*, Vol. 1, No. 2, pp. 60-70, 1982

[41] Torre, V.; Poggio, T.; "A Synaptic Mechanism Possibly Underlying Directional Selectivity to Motion", *Proc. R. Soc. Lond.* Vol. 202 B. pp. 409-416, 1978.

[42] Turner, T., Craig, J.; Gruver, W.A.; "A Microprocessor Architecture for Advanced Robot Control", *Proc. Intl. Conf. on Cybernetics and Society (IEEE)*, October 28-30, 1982, Seattle, Wa., p. 297 (abstract only).

[43] Uicker, J. J.; *On the Dynamic Analysis of Spatial Linkages Using* 4×4 *Matrices*, Ph.D. Thesis, Northwestern U., August 1965.

[44] Waters, R. C.; *Mechanical Arm Control*, M.I.T. Artificial Intelligence Laboratory Memo No. 549, Massachusetts Institute of Technology, Cambridge, Mass., October 1979.

[45] Whitney, D. E.; "The Mathematics of Coordinated Control of Prosthetic Arms and Manipulators",*Journal of Dynamic Systems, Measurement, and Control, Vol. 94*, pp. 303-309, December 1972.

Appendix A — Derivation of the Linear Time Offsets and C

In the following, as in Table III.1, "$Avail(X_{i-1}) = t$" means that variable X_{i-1} is made available to the i^{th} node at time t (on the backward recursion; substitute X_{i+1} on the forward recursion). The abbreviations VA (vector addition), VC (vector cross product), etc., are explained in Table III.1; they denote the time required to perform certain matrix or vector operations.

First we determine the relative delays, or offsets, in variable availability times. We do this by requiring the following implication, for each propagated recursive variable:

$$\begin{aligned} &Avail(X_i) = \max(Avail(X_{i-1}) + \alpha, Avail(Y_{i-1}) + \beta, Avail(Z_{i-1}) + \delta, \ldots) + \gamma \\ \Rightarrow\ &Avail(X_i) = Avail(X_{i-1}) + \alpha + \gamma \end{aligned}$$

This condition amounts to saying that nothing delays the availability of a variable longer than itself. This allows one to infer

$$\max(Avail(X_{i-1}) + \alpha, Avail(Y_{i-1}) + \beta, Avail(Z_{i-1}) + \delta, \ldots) = Avail(X_{i-1}) + \alpha$$

from which immediately follow the inequalities

$$\begin{aligned} Avail(X_{i-1}) + \alpha &\geq Avail(Y_{i-1}) + \beta \\ Avail(X_{i-1}) + \alpha &\geq Avail(Z_{i-1}) + \delta \\ &\ldots \end{aligned}$$

That the set of all such inferrable inequalities is globally satisfiable follows from the ability to globally order the recursion variables

$$X_i \succeq Y_i \succeq Z_i \succeq \ldots$$

where we define

$$\begin{aligned} X_i \succeq Y_i &\text{ iff } Y_i \text{ does not depend on } X_j \text{ for any } j \\ &\text{iff } \partial Y_i / \partial X_j \equiv 0. \end{aligned}$$

In the linear Newton-Euler recursion, for example, we have

$$n_i \succeq f_i \succeq \ddot{p}_i \succeq \dot{\omega}_i \succeq \omega_i$$

and the non-propagated variables could be included in the chain if desired.

By considering starting conditions (initiation of the calculation) one can generate another set of inequalities of the form

$$Avail(X_1) + \alpha' \geq Avail(Y_1) + \beta'$$
$$Avail(X_1) + \alpha' \geq Avail(Z_1) + \delta'$$
$$\cdots$$

Since both sets of inequalities are satisfiable it is possible to reach a point of minimum satisfaction (how dreary... why would one wish to do so?). This is the unique point $(Avail(X_1), Avail(Y_1), Avail(Z_1), \ldots)$ which satisfies both sets of inequalities above, and also minimizes $Avail(X_1)$ for each variable X. This defines the relative offsets which will minimize the total computational time.

On the backward Newton-Euler recursion only ω_i, $\dot{\omega}_i$, and $\ddot{p}_i$ need be considered. This is because $\dot{r}_i$, F_i, and N_i are merely passed directly to forward recursion nodes. From Table III.1,

$$Avail(\dot{\omega}_i) = \max\left(Avail(\omega_{i-1}) + VC + VA,\ Avail(\dot{\omega}_{i-1})\right) + MV + VA$$

Thus for minimum delay $(MV + VA)$ in propagating $\dot{\omega}_i$ we require that

$$\max(Avail(\omega_{i-1}) + VC + VA, Avail(\dot{\omega}_{i-1})) = Avail(\dot{\omega}_{i-1}),$$

hence

$$Avail(\dot{\omega}_{i-1}) \geq Avail(\omega_{i-1}) + VC + VA \qquad (*)$$

Similarly,

$$\begin{aligned}
Avail(\ddot{p}_i) &= \max\big(Avail(\omega_i) + 2VC + VA,\\
&\qquad Avail(\dot{\omega}_i) + VC + VA,\\
&\qquad Avail(\ddot{p}_{i-1}) + MV\big) + VA\\
&= \max\big(Avail(\omega_{i-1}) + MV + 2VC + 2VA,\\
&\qquad (\max(Avail(\dot{\omega}_{i-1}), Avail(\omega_{i-1}) + VC + VA) + MV + VA) + VC + VA,\\
&\qquad Avail(\ddot{p}_{i-1}) + MV\big) + VA\\
&= \max\big(Avail(\omega_{i-1}) + MV + 2VC + 3VA,\\
&\qquad Avail(\dot{\omega}_{i-1}) + MV + VC + 2VA,\\
&\qquad Avail(\ddot{p}_{i-1}) + MV\big) + VA
\end{aligned}$$

And so the minimal delay in propagating $\ddot{p}_i$ requires both

$$\begin{aligned} Avail(\ddot{p}_{i-1}) &\geq Avail(\omega_{i-1}) + 2VC + 3VA \\ Avail(\ddot{p}_{i-1}) &\geq Avail(\dot{\omega}_{i-1}) + VC + 2VA. \end{aligned} \qquad (**)$$

If equality holds in (*) then these two bounds are equivalent.

On the forward recursion only f_i and n_i need be considered.

$$Avail(n_i) = \max(Avail(f_{i+1}) + VC + VA, Avail(n_{i+1})) + MV + VA$$

and so

$$Avail(n_{i+1}) \geq Avail(f_{i+1}) + VC + VA. \qquad (***)$$

Next we determine the constant C by showing when τ_1, the last generalized joint force of the forward recursion, becomes available as output. From Table III.1, assuming all input values become available simultaneously at time $t = 0$,

$$\begin{aligned} Avail(\omega_1) &= MV + VA \\ Avail(\dot{\omega}_1) &= MV + VC + 2VA \\ Avail(\ddot{p}_1) &= \max(Avail(\dot{\omega}_1) + VC + VA, Avail(\omega_1) + 2VC + VA, MV) + VA) \\ &= MV + 2VC + 4VA \end{aligned}$$

Note that these satisfy (*) and (**) so the propagation time is $(MV + VA)$ per node.

$$\begin{aligned} Avail(\omega_n) &= (n-1)(MV + VA) + Avail(\omega_1) \\ &= (n-1)(MV + VA) + MV + VA \\ Avail(\dot{\omega}_n) &= (n-1)(MV + VA) + MV + VC + 2VA \\ Avail(p_n) &= (n-1)(MV + VA) + MV + 2VC + 4VA \\ Avail(\ddot{r}_n) &= \max(Avail(\ddot{p}_n), Avail(\dot{\omega}_n) + VC + VA, Avail(\omega_n) + 2VC + VA) + VA \\ &= (n-1)(MV + VA) + MV + 2VC + 5VA \\ Avail(F_n) &= Avail(\ddot{r}_n) + SV \\ &= (n-1)(MV + VA) + SV + MV + 2VC + 5VA \\ Avail(N_n) &= \max(Avail(\omega_n) + VC, Avail(\dot{\omega}_n)) + MV + VA \\ &= (n-1)(MV + VA) + 2MV + VC + 3VA. \end{aligned}$$

Thereafter, on the forward recursion, (recognizing that $Avail(f_{n+1}) = Avail(n_{n+1}) = 0$),

$$\begin{aligned} Avail(f_n) &= Avail(F_n) + VA \\ &= (n-1)(MV + VA) + SV + MV + 2VC + 6VA \\ Avail(n_n) &= max(Avail(F_n) + VC, Avail(N_n)) + 3VA \\ &= (n-1)(MV + VA) + MV + VC + 6VA \\ &\qquad + \max(SV + 2VC + 2VA, MV) \\ &= (n-1)(MV + VA) + SV + MV + 3VC + 8VA \end{aligned}$$

These two expressions satisfy (***) so propagation occurs at the maximum rate of $(MV + VA)$ per node.

$$\begin{aligned} Avail(n_1) &= 2(n-1)(MV + VA) + SV + MV + 3VC + 8VA \\ Avail(\tau_1) &\leq \max(Avail(f_1), Avail(n_1)) \\ &= 2(n-1)(MV + VA) + SV + MV + 3VC + 8VA \end{aligned}$$

(Actually, $Avail(\tau_1)$ will depend on whether joint i is translational ($= Avail(f_1)$) or rotational ($= Avail(n_1)$), but we assume here rotational, the worst case.)

Assuming a maximally parallel implementation, we would have:

$VA = 1$	$Addn$		(using 3 adders)
$SV = 1$	$Mult$		(using 3 multipliers)
$VC = 1$	$Mult + 1$	$Addn$	(using 6 multipliers and 3 adders)
$MV = 1$	$Mult + 2$	$Addns$	(using 9 multipliers and 3 adders).

So

$$Avail(\tau_1) \leq (2n + 3)\ Mults\ +\ (6n + 7)\ Addns$$

suffice.

Appendix B — Derivation of the Logarithmic Recursive Formulae

NEWTON-EULER BACKWARD RECURSION VARIABLES:

The derivation of the logarithmic combining form for ω_i has been developed in the text. Next, we show that $\dot{\omega}_i$ satisfies the following closed-form formula:

$$\dot{\omega}_i = \sum_{j=0}^{i} W_{j,i}^T \sigma_j (z_{j-1}\ddot{q}_j + \omega_{j-1} \times z_{j-1}\dot{q}_j)$$

as it is a fixed-point of the recursive formula for $\dot{\omega}_i$ in Table I.1:

$$\begin{aligned}\dot{\omega}_i &= A_i^T \Big(\sum_{j=0}^{i-1} W_{j,i-1}^T \sigma_j (z_{j-1}\ddot{q}_j + \omega_{j-1} \times z_{j-1}\dot{q}_j) \\ &\qquad + \sigma_i (z_{i-1}\ddot{q}_i + \omega_{i-1} \times z_{i-1}\dot{q}_i) \Big) \\ &= A_i^T \Big(\dot{\omega}_{i-1} + \sigma_i (z_{i-1}\ddot{q}_i + \omega_{i-1} \times z_{i-1}\dot{q}_i) \Big)\end{aligned}$$

As in the case of ω_i, we take $\dot{\omega}_0 = A_0^T z_{(-1)}\ddot{q}_0$. Most applications of interest will have $\dot{\omega}_0 = \ddot{q}_0 = 0$.

In order to match at $a = 0$ and $b = i$,

$$\begin{aligned}\dot{\omega}_{a,b} &\equiv \sum_{j=a}^{b} W_{j,b}^T \sigma_j (z_{j-1}\ddot{q}_j + \omega_{a,(j-1)} \times z_{j-1}\dot{q}_j) \\ &= \sum_{j=a}^{k} \left(W_{j,k} W_{(k+1),b}\right)^T \sigma_j (z_{j-1}\ddot{q}_j + \omega_{a,(j-1)} \times z_{j-1}\dot{q}_j) \\ &\qquad + \sum_{j=k+1}^{b} W_{j,b}^T \sigma_j \Big(z_{j-1}\ddot{q}_j + (W_{(k+1),(j-1)}^T \omega_{a,k} + \omega_{(k+1),(j-1)}) \times z_{j-1}\dot{q}_j \Big) \\ &= W_{(k+1),b}^T \dot{\omega}_{a,k} + \dot{\omega}_{(k+1),b} + \sum_{j=k+1}^{b} W_{j,b}^T \sigma_j \Big((W_{(k+1),(j-1)}^T \omega_{a,k}) \times z_{j-1}\dot{q}_j \Big) \\ &= W_{(k+1),b}^T \dot{\omega}_{a,k} + (W_{(k+1),b}^T \omega_{a,k}) \times \omega_{(k+1),b} + \dot{\omega}_{(k+1),b}\end{aligned}$$

which is the combining form for $\dot{\omega}_{a,b}$.

To derive the combining form for $\ddot{p}_{a,b}$, it is necessary to create the auxiliary variables

$$\begin{aligned}
Q_{a,b} &\equiv \sum_{j=a}^{b} W_{j,b}^T \bar{\sigma}_j z_{j-1} \dot{q}_j \\
&= W_{k+1,b}^T Q_{a,k} + Q_{(k+1),b} \\
R_{a,b} &\equiv \sum_{j=a}^{b} W_{j+1,b}^T p_j^* \\
&= W_{k+1,b}^T R_{a,k} + R_{(k+1),b} \\
S_{a,b} &\equiv \sum_{j=a}^{b} W_{j+1,b}^T (\omega_{a,j} \times p_j^*) \\
&= W_{k+1,b}^T S_{a,k} + \sum_{j=k+1}^{b} W_{j+1,b}^T \left((W_{k+1,j}^T \omega_{a,k} + \omega_{k+1,j}) \times p_j^* \right) \\
&= W_{k+1,b}^T S_{a,k} + (W_{k+1,b}^T \omega_{a,k}) \times R_{(k+1),b} + S_{(k+1),b}
\end{aligned}$$

Next we show that $\ddot{p}_i$ satisfies the closed-form expression

$$\begin{aligned}
\ddot{p}_i = W_{1,i}^T \ddot{p}_0 + \sum_{j=1}^{i} W_{j+1,i}^T \Big(& \dot{\omega}_j \times p_j^* + \omega_j \times (\omega_j \times p_j^*) \\
& + \bar{\sigma}_j (A_j^T z_{j-1} \ddot{q}_j + 2\omega_j \times A_j^T z_{j-1} \dot{q}_j) \Big)
\end{aligned}$$

This is a fixed point of the recursive formula for $\ddot{p}_i$ as shown

$$\begin{aligned}
\ddot{p}_i = A_i^T \Bigg(& W_{1,i-1}^T \ddot{p}_0 + \sum_{j=1}^{i-1} W_{j+1,i-1}^T \Big(\dot{\omega}_j \times p_j^* + \omega_j \times (\omega_j \times p_j^*) \\
& + \bar{\sigma}_j (A_j^T z_{j-1} \ddot{q}_j + 2\omega_j \times A_j^T z_{j-1} \dot{q}_j) \Big) \Bigg) \\
& + W_{i+1,i}^T \Big(\dot{\omega}_i \times p_i^* + \omega_i \times (\omega_i \times p_i^*) \\
& + \bar{\sigma}_i (A_i^T z_{i-1} \ddot{q}_i + 2\omega_i \times A_i^T z_{i-1} \dot{q}_i) \Big) \\
= A_i^T \ddot{p}_{i-1} & + \dot{\omega}_i \times p_i^* + \omega_i \times (\omega_i \times p_i^*) \\
& + \bar{\sigma}_i (A_i^T z_{i-1} \ddot{q}_i + 2\omega_i \times A_i^T z_{i-1} \dot{q}_i)
\end{aligned}$$

where $\ddot{p}_0$ is the acceleration of the base. Typically this is the acceleration due to gravity at the site. If one took $\omega_0 \neq 0$ above then $\ddot{p}_0$ may also include a term for $\omega_0 \times (\omega_0 \times p_0^*)$, where

p_0^* is a vector from the Earth's center $O_{(-1)}$ to the site; this accounts for the centripetal acceleration arising from the rotation of the Earth. If one accounts for the gravitational acceleration (g) by taking $\ddot{p}_0^g = g$ at the base, else $\ddot{p}_i^g = 0$ for $i \neq 0$, then the formula may be equivalently re-written with greater clarity (covering both cases $\omega_0 = 0$ and $\omega_0 \neq 0$) as

$$\ddot{p}_i = \sum_{j=0}^{i} W_{j+1,i}^T \Big(\ddot{p}_j^g + \dot{\omega}_j \times p_j^* + \omega_j \times (\omega_j \times p_j^*) + \bar{\sigma}_j (A_j^T z_{j-1} \ddot{q}_j + 2\omega_j \times A_j^T z_{j-1} \dot{q}_j) \Big)$$

This we will take to be the defining closed-form non-recursive formula for $\ddot{p}_i$. The term involving $\ddot{p}_j^g$ is a technical artifice to account for $\ddot{p}_0$ cleanly, and will vanish in the combining form.

The demonstration of the combining form of $\ddot{p}_{a,b}$ will require the vector identity $a \times (b \times c) + (b \times a) \times c = b \times (a \times c)$. In order to match at $a = 0$ and $b = i$, take

$$\ddot{p}_{a,b} \equiv \sum_{j=a}^{b} W_{j+1,b}^T \left(\ddot{p}_j^g + \dot{\omega}_{a,j} \times p_j^* + \omega_{a,j} \times (\omega_{a,j} \times p_j^*) + \bar{\sigma}_j (A_j^T z_{j-1} \ddot{q}_j + 2\omega_{a,j} \times A_j^T z_{j-1} \dot{q}_j) \right)$$

$$= W_{k+1,b}^T \sum_{j=a}^{k} W_{j+1,k}^T \left(\ddot{p}_j^g + \dot{\omega}_{a,j} \times p_j^* + \omega_{a,j} \times (\omega_{a,j} \times p_j^*) \right.$$

$$\left. + \bar{\sigma}_j (A_j^T z_{j-1} \ddot{q}_j + 2\omega_{a,j} \times A_j^T z_{j-1} \dot{q}_j) \right)$$

$$+ \sum_{j=k+1}^{b} W_{j+1,b}^T \left(\ddot{p}_j^g + \left(W_{k+1,j}^T \dot{\omega}_{a,k} + (W_{k+1,j}^T \omega_{a,k}) \times \omega_{(k+1),j} + \dot{\omega}_{(k+1),j} \right) \times p_j^* \right.$$

$$+ (W_{k+1,j}^T \omega_{a,k} + \omega_{(k+1),j}) \times \left((W_{k+1,j}^T \omega_{a,k} + \omega_{(k+1),j}) \times p_j^* \right)$$

$$\left. + \bar{\sigma}_j \left(A_j^T z_{j-1} \ddot{q}_j + 2(W_{k+1,j}^T \omega_{a,k} + \omega_{k+1,j}) \times A_j^T z_{j-1} \dot{q}_j \right) \right)$$

$$= W_{k+1,b}^T \ddot{p}_{a,k} + \ddot{p}_{(k+1),b} + (W_{k+1,b}^T \dot{\omega}_{a,k}) \times \sum_{j=k+1}^{b} W_{j+1,b}^T p_j^*$$

$$+ (W_{k+1,b}^T \omega_{a,k}) \times \left((W_{k+1,b}^T \omega_{a,k}) \times \sum_{j=k+1}^{b} W_{j+1,b}^T p_j^* \right.$$

$$\left. + 2 \sum_{j=k+1}^{b} \left(W_{j+1,b}^T (\omega_{(k+1),j} \times p_j^*) + \bar{\sigma}_j W_{j+1,b}^T (A_j^T z_{j-1} \dot{q}_j) \right) \right)$$

$$\ddot{p}_{a,b} = (W_{k+1,b}^T \ddot{p}_{a,k}) + \ddot{p}_{(k+1),b} + (W_{k+1,b}^T \dot{\omega}_{a,k}) \times R_{(k+1),b}$$

$$+ (W_{k+1,b}^T \omega_{a,k}) \times \left((W_{k+1,b}^T \omega_{a,k}) \times R_{(k+1),b} + 2(S_{(k+1),b} + Q_{k+1,b}) \right)$$

NEWTON-EULER FORWARD RECURSION VARIABLES

As noted in the text, on the Newton-Euler forward recursion the coordinate matrix products of interest will be $W_{a+1,k+1}$ instead of $W_{k+1,b}^T$.

Noting that the numbering runs backward, we see that f_i satisfies the non-recursive formula

$$f_i = \sum_{j=i}^{n} W_{i+1,j} F_j$$

$$= A_{i+1} \sum_{j=i+1}^{n} W_{i+2,j} F_j + F_i$$

$$= A_{i+1} f_{i+1} + F_i$$

To match at $a = i$ and $b = n$,

$$\begin{aligned} f_{a,b} &\equiv \sum_{j=a}^{b} W_{a+1,j} F_j \\ &= \sum_{j=a}^{k} W_{a+1,j} F_j + W_{a+1,k+1} \sum_{j=k+1}^{b} W_{k+2,j} F_j \\ &= f_{a,k} + W_{a+1,k+1} f_{(k+1),b} \end{aligned}$$

If desired, forces and torques applied by the environment to the manipulator tip may be incorporated in a fashion similar to incorporating the acceleration of the base in the discussion of $\ddot{p}_{i,i}$. This will not change the displayed combining forms.

Similarly, n_i satisfies the non-recursive formula

$$\begin{aligned} n_i &= \sum_{j=i}^{n} W_{i+1,j} \left(N_j + s_j^* \times F_j + p_j^* \times (A_{j+1} f_{j+1}) \right) \\ &= N_i + s_i^* \times F_i + p_i^* \times (A_{i+1} f_{i+1}) \\ &\qquad + A_{i+1} \sum_{j=i+1}^{n} W_{i+2,j} \left(N_j + s_j^* \times F_j + p_j^* \times (A_{j+1} f_{j+1}) \right) \\ &= N_i + s_i^* \times F_i + p_i^* \times (A_{i+1} f_{i+1}) + A_{i+1} n_{i+1} \end{aligned}$$

To match at $a = i$ and $b = n$ we must have

$$\begin{aligned} n_{a,b} &\equiv \sum_{j=a}^{b} W_{a+1,j} \left(N_j + s_j^* \times F_j + p_j^* \times (A_{j+1} f_{(j+1),b}) \right) \\ &= \sum_{j=a}^{k} W_{a+1,j} \left(N_j + s_j^* \times F_j + p_j^* \times (A_{j+1} (f_{j+1,k} + W_{j+2,k+1} f_{k+1,b})) \right) \\ &\qquad + W_{a+1,k+1} \sum_{j=k+1}^{b} W_{k+2,j} \left(N_j + s_j^* \times F_j + p_j^* \times (A_{j+1} f_{(j+1),b}) \right) \\ &= n_{a,k} + W_{a+1,k+1} n_{k+1,b} + \sum_{j=a}^{k} W_{a+1,j} \left(p_j^* \times (W_{j+1,k+1} f_{k+1,b}) \right) \\ &= n_{a,k} + W_{a+1,k+1} \left((A_{k+1}^T R_{a,k}) \times f_{k+1,b} + n_{k+1,b} \right) \end{aligned}$$

Appendix C — Derivation of the Logarithmic Time Offsets and C

These may be derived from Table V.1 by inspection, by applying the rule that to insure minimum delay, the maximum delay of any variable must be caused by a data dependency on that variable.

$$
\begin{aligned}
Avail(\omega_{x,y}) &\geq Avail(W_{x,y}) \\
Avail(\dot{\omega}_{x,y}) &\geq Avail(W_{x,y}) + VC + VA \\
Avail(\dot{\omega}_{x,y}) &\geq Avail(\omega_{x,y}) + VC + VA \\
Avail(R_{x,y}) &\geq Avail(W_{x,y}) \\
Avail(S_{x,y}) &\geq Avail(W_{x,y}) + VC + VA \qquad (*) \\
Avail(S_{x,y}) &\geq Avail(\omega_{x,y}) + VC + VA \\
Avail(S_{x,y}) + MV &\geq Avail(R_{x,y}) + VC + VA \\
Avail(Q_{x,y}) &\geq Avail(W_{x,y}) \\
Avail(\ddot{p}_{x,y}) &\geq Avail(W_{x,y}) + 2VC + 3VA \\
Avail(\ddot{p}_{x,y}) &\geq Avail(\omega_{x,y}) + 2VC + 3VA \\
Avail(\ddot{p}_{x,y}) &\geq Avail(\dot{\omega}_{x,y}) + VC + 2VA \\
Avail(\ddot{p}_{x,y}) + MV &\geq Avail(R_{x,y}) + 2VC + 3VA \\
Avail(\ddot{p}_{x,y}) + MV &\geq Avail(S_{x,y}) + VC + 4VA + SV \qquad (*) \\
Avail(\ddot{p}_{x,y}) + MV &\geq Avail(Q_{x,y}) + VC + 4VA + SV \\
Avail(\ddot{p}_{x,y}) + MV &\geq Avail(W_{x,y}) + 2VC + 5VA + SV \qquad (from\ (*)\ above) \\
\\
Avail(n_{x,y}) &\geq Avail(f_{x,y}) + VC + VA
\end{aligned}
$$

The delay conditions established, actual timing can be generated. From Table IV.1 we extract the $a = b$ case; Table V.1 covers $a \neq b$.

$$
\begin{aligned}
Avail(W_{a,a}) &= 0 \\
Avail(\omega_{a,a}) &= MV \\
Avail(\dot{\omega}_{a,a}) &= MV \qquad (*) \\
Avail(Q_{a,a}) &= MV \\
Avail(R_{a,a}) &= 0 \\
Avail(S_{a,a}) &= MV + VC. \qquad (*)
\end{aligned}
$$

However, the equations marked (*) fail to satisfy the delay conditions and so must be revised to

$$
\begin{aligned}
Avail(\dot{\omega}_{a,a}) &= MV + VC + VA \\
Avail(S_{a,a}) &= MV + VC + VA.
\end{aligned}
$$

Analysis of $Avail(\ddot{p}_{a,a})$ is less obvious, but proceeds as follows (assuming $VC \geq SV$ and $VC \geq VA$)

$$\begin{aligned}
Avail(\ddot{p}_a^0) &= 0 \\
Avail(\bar{\sigma}_a A_a^T z_{a-1} \ddot{q}_a) &= MV \\
Avail(\ddot{p}_a^0 + \bar{\sigma}_a A_a^T z_{a-1} \ddot{q}_a) &= MV + VA \\
Avail(\omega_{a,a} \times (\omega_{a,a} \times p_a^*)) &= MV + 2VC \\
Avail(\ddot{p}_a^0 + \bar{\sigma}_a A_a^T z_{a-1} \ddot{q}_a + \omega_{a,a} \times (\omega_{a,a} \times p_a^*)) &= MV + 2VC + VA \\
Avail(\dot{\omega}_{a,a} \times p_a^*) &= MV + VC \\
Avail(2\omega_{a,a} \times Q_{a,a}) &= MV + SV + VC \\
Avail(\dot{\omega}_{a,a} \times p_a^* + 2\omega_{a,a} \times Q_{a,a}) &= MV + SV + VC + VA \\
Avail(\ddot{p}_{a,a}) &= MV + 2VC + 2VA \qquad (*) \\
Avail(\ddot{p}_{a,a}) &= MV + 2VC + 3VA
\end{aligned}$$

where the last line is added so that $\ddot{p}_{a,a}$ satisfies the delay conditions (assuming $MV \geq SV + 2VA$).

Since these satisfy the minimum delay conditions, propagation occurs at a rate of $(MV + VA)$ per node. It can readily be seen that, in general,

$$Avail(X_{a,b}) = Avail(X_{a,a}) + \lceil \log_2(b - a + 1) \rceil (MV + VA).$$

Thus in particular, if X_i is the linear recursive variable corresponding to $X_{a,b}$, then $X_i = X_{0,i}$ so

$$\begin{aligned}
Avail(X_i) &= Avail(X_{0,i}) \\
&= Avail(X_{a,a}) + \lceil \log_2(i + 1) \rceil (MV + VA).
\end{aligned}$$

Hence,

$$\begin{aligned}
Avail(\omega_{0,i}) &= Avail(\omega_{a,a}) + \lceil \log_2(i + 1) \rceil (MV + VA) \\
Avail(\dot{\omega}_{0,i}) &= Avail(\dot{\omega}_{a,a}) + \lceil \log_2(i + 1) \rceil (MV + VA) \\
Avail(\ddot{p}_{0,i}) &= Avail(\ddot{p}_{a,a}) + \lceil \log_2(i + 1) \rceil (MV + VA) \\
Avail(\ddot{r}_{0,i}) &= \max(Avail(\ddot{p}_{0,i}), Avail(\dot{\omega}_{0,i}) + VC + VA, Avail(\omega_{0,i}) + 2VC + VA) + VA \\
Avail(F_{0,i}) &= Avail(\ddot{r}_{0,i}) + SV \\
&= MV + SV + 2VC + 4VA + \lceil \log_2(i + 1) \rceil (MV + VA) \\
Avail(N_{0,i}) &= \max(Avail(\omega_{0,i}) + VC, Avail(\dot{\omega}_{0,i})) + MV + VA \\
&= 2MV + VC + 2VA + \lceil \log_2(i + 1) \rceil (MV + VA)
\end{aligned}$$

Thereafter, on the forward recursion (from Table IV.1),

$$\begin{aligned}
Avail(f_{n,n}) &= Avail(F_{0,n}) \\
&= MV + SV + 2VC + 4VA + \lceil \log_2(n+1) \rceil (MV + VA) \\
Avail(n_{n,n}) &= \max(Avail(F_{0,n}) + VC, Avail(N_{0,n})) + VA \\
&= MV + VC + 3VA \\
&\qquad + \max(MV, SV + 2VC + 2VA) + \lceil \log_2(n+1) \rceil (MV + VA)
\end{aligned}$$

which satisfies the delay conditions. Propagation therefore occurs at the maximum rate and

$$\begin{aligned}
Avail(n_{n,0}) &= MV + VC + 3VA \\
&\qquad + \max(MV, SV + 2VC + 2VA) + 2\lceil \log_2(n+1) \rceil (MV + VA) \\
Avail(\tau_0) &= MV + VC + 3VA \\
&\qquad + \max(MV, SV + 2VC + 2VA) + 2\lceil \log_2(n+1) \rceil (MV + VA) \\
&= 2\lceil \log_2(n+1) \rceil (MV + VA) + MV + SV + 3VC + 5VA \\
&= 2\lceil \log_2(n+1) \rceil (MV + VA) + 5\ \textit{Mults} + 10\ \textit{Addns} \\
&\geq Avail(\tau_1)
\end{aligned}$$

assuming again a maximally parallel system.

Appendix D — Unification of Logarithmic $a = b$ and $a \neq b$ Cases

By the following technical artifice we can make the $a = b$ case look like $a \neq b$.

$$
\begin{aligned}
W_{a,k} &= I \\
W_{k+1,b} &= A_a \\
\omega_{a,k} &= \sigma_a z_{a-1} \dot{q}_a \\
\omega_{k+1,b} &= 0 \\
\dot{\omega}_{a,k} &= \sigma_a z_{a-1} \ddot{q}_a \\
\dot{\omega}_{k+1,b} &= 0 \\
Q_{a,k} &= \bar{\sigma}_a z_{a-1} \dot{q}_a \\
Q_{k+1,b} &= 0 \\
R_{a,k} &= 0 \\
R_{k+1,b} &= p_a^* \\
S_{a,k} &= 0 \\
S_{k+1,b} &= 0 \\
\ddot{p}_{a,k} &= \bar{\sigma}_a a_{a-1} \ddot{q}_a \\
\ddot{p}_{k+1,b} &= \ddot{p}_a^g
\end{aligned}
$$

$Q_{a,a}$ is substituted for $Q_{k+1,b}$ in $\ddot{p}_{a,a}$

$$
\begin{aligned}
W_{a+1,k+1} &= I \\
W_{k+2,b+1} &= A_{a+1} \\
f_{a,k} &= F_a \\
f_{k+1,b} &= 0 \\
n_{a,k} &= N_a \\
n_{k+1,b} &= s_a^* \times F_a.
\end{aligned}
$$

Now following two applications of the $(n/2)$ processor nodes to the n groups of input data we have $X_{a,a}$ as required, and similarly on the forward recursion.


DAT
FILM